НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет Інформатики та обчислювальної техніки

Обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____Сергій СТІРЕНКО
«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Система безпечного завантаження та виконання коду на сервері»

Виконав:
студент IV курсу, групи ІО-64
Ханін Максим Євгенійович _____

Керівник:
доктор технічних наук, професор
Новотарський Михайло Анатолійович _____

Консультант з нормоконтролю:
доктор технічних наук, професор
Сімоненко Валерій Павлович _____

Рецензент:
кандидат технічних наук, доцент
Орлова Марія Миколаївна _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
«___»_____20__ р.

ЗАВДАННЯ

на дипломну роботу студенту
Ханіну Максиму Євгенійовичу

1. Тема роботи «Система безпечного завантаження та виконання коду на сервері», керівник роботи Новотарський Михайло Анатолійович, доктор технічних наук, професор, затверджені наказом по університету від «07» травня 2020 р. № 1081-с

2. Термін подання студентом роботи _____

3. Вихідні дані до роботи: технічна документація, теоретичні дані;

4. Зміст роботи: методи запуску коду на сервері, технологія Docker, структура системи, результати розробки;

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): принципова схема алгоритму завантаження та виконання коду на сервері, структурна схема системи завантаження та виконання коду на сервері, функціональна схема класів системи завантаження та виконання коду на сервері.

6. Консультанти розділів роботи*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	д.т.н, проф. Сімоненко В. П.		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Затвердження теми роботи	01.09.2019	
2	Вивчення та аналіз завдання	20.01.2020 – 09.02.2020	
3	Розробка загальної структури системи	10.02.2020 – 23.02.2020	
4	Програмна реалізація системи	24.02.2020 – 19.04.2020	
5	Тестування та виправлення помилок	20.04.2020 – 03.05.2020	
6	Оформлення пояснювальної записки	04.05.2020 – 25.05.2020	
7	Передзахист	26.05.2020	
8	Захист	15.06.2020	

Студент Максим ХАНІН

Керівник Михайло НОВОТАРСЬКИЙ

Анотація

Дана дипломна робота присвячена розробці сервісу для безпечного завантаження та виконання коду на сервері. Даний сервіс допомагає уникнути проблем при виконанні чужого коду на своїй системі. Програмний код та результат його виконання можна передивлятися прямо на сайті чи надсилати запити самостійно. Система дає змогу перевірити роботу програми без завантаження додаткового програмного забезпечення. Адміністратор також має змогу змінити деякі параметри налаштування самостійно, а саме: ліміт оперативної пам’яті чи максимальний час виконання програми. Таким чином, користувач може перевірити чужий код та не завдати шкоди свої системі та серверу через те, що весь код завантажується і виконується в контейнерах.

Аннотация

Данная дипломная работа посвящена разработке сервиса для безопасной загрузки и выполнения кода на сервере. Данный сервис помогает избежать проблем при выполнении чужого кода на своей системе. Программный код и результат его выполнения можно просматривать прямо на сайте или отправлять запросы самостоятельно. Система позволяет проверить работу программы без загрузки дополнительного программного обеспечения. Администратор также имеет возможность изменить некоторые параметры настройки самостоятельно, а именно: лимит оперативной памяти или максимальное время выполнения программы. Таким образом, пользователь может проверить чужой код и не нанести вреда своей системе и серверу из-за того, что весь код загружается и выполняется в контейнерах.

Annotation

This diploma thesis is devoted to the development of a service for secure downloading and execution of code on the server. This service helps to avoid problems when executing untrusted code on your system. You can view the program code and the result of its execution directly on the site or send requests yourself. The system allows you to perform tasks and test the program without downloading additional software. The administrator also has the ability to change some of the settings themselves, such as memory limit or maximum program execution time. In this way, the user can check untrusted code and not harm their system and server because all the code is downloaded and executed in containers.

Опис альбому
до дипломного проєкту
на тему: «Система безпечного завантаження та виконання коду на сервері»

Київ – 2020 року

№ рядка	Формат	Позначення	Найменування	Кількість	Примітка
1			Завдання на дипломний проект	2	
2					
3	А4	*ІАЛЦ.467100.001 ОА*	Опис альбому	1	
4					
5	А4	*ІАЛЦ.467100.002 ТЗ*	Технічне завдання	3	
6					
7	А4	*ІАЛЦ.467100.003 ПЗ*	Пояснювальна записка	55	
8					
9	А4	*ІАЛЦ.467100.004 Д1*	Додаток А	1	
10					
11	А4	*ІАЛЦ.467100.005 Д2*	Додаток Б	1	
12					
13	А4	*ІАЛЦ.467100.006 Д3*	Додаток В	1	
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					
26					
27					

					ІАЛЦ.467100.001 ОА		
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		*Ханін М. Є.*			*Система безпечного завантаження та виконання коду на сервері* *Опис альбому*	*Літ.*	*Аркуш*	*Аркушів*
Перевір.		*Новотарський М. А.*					*1*	*1*
						НТУУ «КПІ», ФІОТ ІО-64		
Н. Контр.		*Сімоненко В. П.*						
Затверд.								

Технічне завдання
до дипломного проєкту
на тему: «Система безпечного завантаження та виконання коду на сервері»

Київ – 2020 року

Технічне завдання до дипломної роботи

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання розповсюджується на розробку системи для безпечного завантаження та виконання коду на сервері.

Область застосування: перевірка програмного коду, без використання додаткового програмного забезпечення.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на створення онлайн системи для завантаження і виконання коду на сервері.

3. МЕТА РОЗРОБКИ

Метою даної розробки є створення системи для завантаження і виконання коду на сервері.

4. ДЖЕРЕЛА РОЗРОБКИ

Основними джерелами для розробки слугують публікації на дану тему в Інтернеті, документації існуючих програм з функціоналом для рішення цього питання.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розроблюваної системи

- Безпечне виконання коду на сервері
- Можливість вводити та запускати програмний код онлайн
- Можливість обирати мову програмування згідно з дисципліною
- Вивід результатів компіляції та виконання програми

5.2. Вимоги до програмного забезпечення

- Google Chrome, або інший аналогічний браузер

5.3. Вимоги до апаратного забезпечення

- Комп’ютер з підключення до мережі Інтернет

6. ЕТАПИ РОЗРОБКИ

Вивчення літератури та інших матеріалів	20.01.2020
Складання та узгодження технічного завдання	03.02.2020
Аналіз доступних технологій	10.02.2020
Розробка модулів розроблюваної системи	24.02.2020
Тестування системи	20.04.2020
Виправлення помилок	27.04.2020
Оформлення документації дипломної роботи	04.05.2020

Пояснювальна записка
до дипломного проєкту
на тему: «Система безпечного завантаження та виконання коду на сервері»

Київ – 2020 року

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

IDE	Integrated development environment
LXC	Linux Containers
LXD	Linux Containers daemon
REST API	Representational State Transfer Application Programming Interface
PaaS	Platform as a service
UnionFS	Union File System

ВСТУП

В сучасному світі все більше технологій переходить в он-лайн мережу. Більшість компаній мають налагоджену систему роботи працівників через мережу Інтернет, при чому та робота, яка потребувала встановлення певного програмного забезпечення, нині частково може виконуватися тільки за допомогою браузера і підключення до мережі. Всі необхідні операції такої роботи виконуються на он-лайн сервісі, а розрахунки на спеціальних виділених серверах, а робітник, відповідно, отримує результати виконання.

Не дивно, що для програмування, тепер, не завжди необхідні встановлені на комп'ютер IDE, наприклад, якщо під рукою немає комп'ютера з необхідним для вас компілятором, а треба швидко перевірити свою програму, то сучасні технології надають доступ до онлайн-компіляторів, які в розумних межах надають таку можливість. Структура таких компіляторів складається з інтерфейсу користувача та серверної частини, користувач обирає необхідну мову та вводить код своєї програми, які відправляються на сервер, на сервері ж виконуються всі розрахунки, а результати перенаправляються на сервер.

Головною задачею при створенні таких сервісів є захист сервера. Оскільки він приймає програмний код, який може нашкодити, як серверу, так і загалом всій системі. Проводити перевірку кожної програми неможливо, а якщо допустити злам системи то це призведе до фатальних наслідків. Так можна втратити дані користувачів, що завадить, або зупинить їх подальшу роботу в системі. Також, код може викликати перевантаження системи, що не є занадто критичним, але може завадити використання системи на певний час. Ну і нарешті система може бути зламана і неможливістю до відновлення.

Постає завдання у виборі технології для безпечного запуску та виконанні коду на сервері. Доречним було б використання певних обмежень на сервері, наприклад часових або ресурсних, а також використання системи ізоляції виконуваної програми, що надає змогу запускати код, наприклад, на

виділеній віртуальній машині. При виконанні програми та повернені даних користувачу, файли цієї програми на сервері можуть знищуватися, для запобігання перевантаження та засмічування системи.

В даній роботі описується методика рішення поставлених завдань, опис існуючих систем для запуску коду на сервері, їх порівняльна характеристика, та створення системи на основі однієї з таких технологій.

РОЗДІЛ 1

МЕТОДИ ЗАПУСКУ КОДУ НА СЕРВЕРІ

Роботу будь-якого он-лайн компілятора, тобто запуску коду на сервері а не на власному комп'ютері можна узагальнити, всі вони мають певну однакову систему роботи, як на рис.1.1. Щоб використовувати таку систему, необхідне тільки підключення до мережі Інтернет.

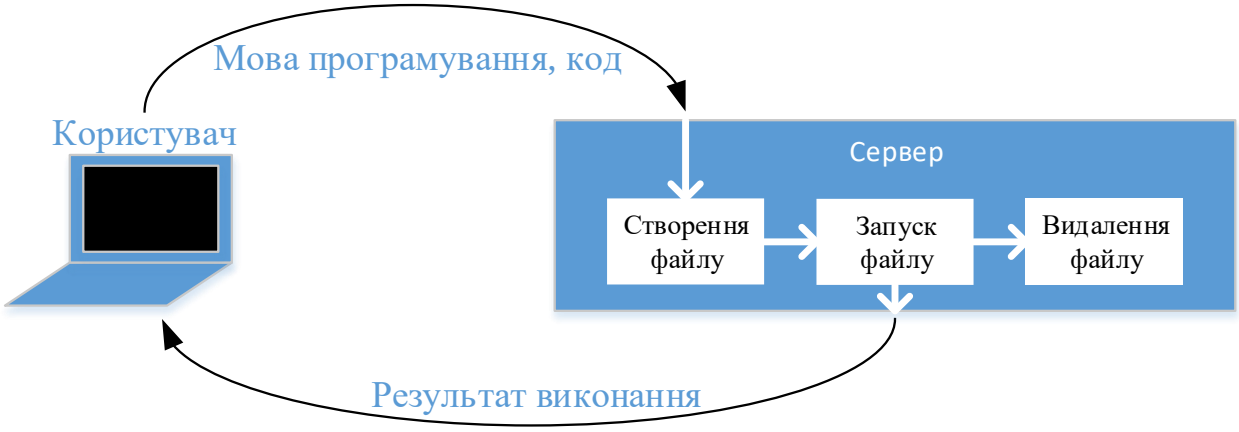


Рис.1.1. Схема роботи он-лайн компілятора

Користувач системи повинен обрати мову компілятора на якій він хоче програмувати, та записати код програми яку він хоче запустити, у разі ж, якщо система працює тільки з однією мовою, то тільки код. Далі ці данні потрапляють на сервер, де зберігаються у вигляді файлу з розширенням для зазначеної мови, та змістом відповідним до наданого коду. На сервер мають бути встановлені всі компілятори наданих на сервісі мов, з усіма розширеннями зазначеними на сайті. Далі сервером визначається необхідний компілятор і створений файл запускається через консоль серверу за допомогою цього компілятора, після чого отримані результати записуються в файл результату і цей файл відправляється назад користувачу. Після цих операцій початковий файл, та файл з результатом знищуються.

Такий загальний підхід використовується у всіх системах для запуску коду он-лайн. Він є небезпечним для власника системи і може бути ненадійним для користувача.

Проблемою такого підходу є самі данні які подаються на сервер. Програма яку отримує сервер може містити будь-який код, тому не виключено, що він може впливати на операційну систему, або на конкретні данні. Такий вплив може бути як навмисним, так і ненавмисний, але результат все одно може викликати серйозні проблеми як для користувача так і для власника сервісу. Всі ці програми не перевіряються на наявність такого коду, та це і не можливо – перевірити всі можливі варіанти. Загальними рішеннями проблеми є використання певних обмежень, але й вони не надають повної безпеки, тому найефективнішим є ізоляція користувацьких програм від основної системи серверу. Такий принцип в загальному ділиться на технології віртуалізації та більш конкретного її різновиду – конвеєризації.

1.1. Віртуалізація

На одному комп’ютері існує можливість запускати одразу кілька операційних систем, що будуть працювати одночасно, такі системи називають віртуальними машинами. Загалом це повноцінне розбиття системних та апаратних ресурсів на кілька емуляторів які повністю копіюють роботу комп’ютерної системи, що і дозволяє встановити будь-яку операційну систему на такі емулятори. Ці емулятори працюють окремо один від одного і залежать тільки від хостової системи, тобто така ізольованість запобігає впливу системи одного емулятора на інший.

Основними мінусами такого підходу є навантаження, яке викликає емулятор при спільній роботі з основною системою та іноді з іншими емуляторами, а також часткова, або, інколи, повна неможливість розподілу ресурсів між емуляторами.

1.2. Контейнеризація

Роботу ядра операційної системи можна розділити на кілька частин простору користувача – зони, при цьому ці частини будуть ізольовані одна відносно іншої, що завадить будь-якому впливу однієї зони на іншу. Такі окремі зони називаються контейнерами, а самий метод контейнеризацією, або віртуалізацією операційної системи. Цей метод дозволяє створити кілька серверів-контейнерів на одному пристрої, які не будуть сильно відрізнятися від реального окремого сервера для звичайного користувача.

Мінусом такої віртуалізації є те, що контейнерам не можна надавати будь-яку операційну систему, всі контейнери будуть підтримувати операційну систему ядра, яке має основна операційна система. При цьому саме ядро дозволяє підтримувати ізольованість зон одна від одної. Такий принцип дозволяє без зайвих ресурсних витрат на емуляцію, запускати декілька екземплярів операційних систем.

Існує кілька технологій які здійснюють контейнеризацію, при чому загалом вони розділені на часткові, для створення певних ізольованих сервісів з мінімальними можливостями оточення, та загальні, які створюють практично копію операційної системи хоста.

1.3. Переваги контейнеризації

Основною і загальною перевагою контейнеризації над застосуванням віртуальних машин є використання ресурсів сервера. Через емуляції роботи обладнання, а не тільки операційної системи, віртуальна машина використовує велику кількість ресурсів таки як оперативна пам’ять, пристроїв зберігання та інших. При цьому розміри віртуальних машин можуть бути занадто великими, що перешкоджає їх збільшенню на сервері. Загальне використання ресурсів сервера може бути нескоординоване між кількома емуляторами, що призведе до великих затримок, або збоїв системи в цілому. В той же час контейнери можуть використовувати мінімум

ресурсів, оскільки для деяких, навіть не має необхідності у повній симуляції операційної системи, а тільки певних її аспектів, або додатків, для отримання потрібних від конкретного контейнера результатів.

В результаті якщо порівнювати контейнери і віртуальні машини, то сервер побудований на контейнерній технології зможе одночасно виконувати в кілька разів більше операцій ніж при використанні загальних віртуальних машин.

1.4. LXC та LXD

LXC – це набір інструментів для локалізації ядра для Linux подібних систем. За допомогою даного інтерфейсу, користувачу надаються можливості для створення та керування контейнерами. Такі контейнери можуть бути як системними так і для додатків. [1]

Оскільки контейнери LXC намагаються максимально наблизитися до системи, то їх часто називають середнім між вбудованим в Unix-подібні системи chroot та віртуальною машиною. Такі види контейнерів називають інфраструктурними машинами. Тобто контейнери мають майже повноцінну Linux систему, і керування відбувається з ними як при наявності повноцінного фізичного обладнання. Ці фактори призводять до певних складнощів у використанні даного типу контейнерів. Для користувача така система може показатися досить складною, а також вона потребує певних початкових знань для роботи з нею. Ще одних труднощів завдає проблема з неможливістю зміни певних налаштувань безпеки, або необхідністю постійної їх зміни вручну. Попри це, LXC надає користувачу великий спектр інструментарію, як низькорівневого так і спеціалізованих бібліотек для роботи з контейнерами, зворотну сумісність зі старими методами та контейнерами, що все одно не вирішує вищезазначених проблем із роботою з таким типом контейнерів.

LXD – це по суті новий етап розвитку LXC, а точніше його альтернатива, що побудована на основі LXC, але надає більш зручний у користуванні інтерфейс, що дозволяє керувати контейнерами, можливістю керувати контейнерами через мережу за допомогою REST API, і являє собою менеджер для системних контейнерів на основі LXC.

Основними особливостями системи LXD є:

- Можливість обмеження ресурсів, наявність непривілейованих контейнерів та інше;
- Можливість створювати як кілька контейнерів на одній машині, так і тисячі зв’язаних обчислювальних вузлів:
- Підтримка більшості дистрибутивів Linux;
- Підтримка cross-host container;
- Розвинутий контроль за ресурсами системи, такими як пам’ять процесори, ресурси ядра та інші;
- Можливість створення на настроювання мережевих тунелів, мостів між кількома хостами;

1.4.1. Образи

Образи Linux дистрибутивів являються основою LXD контейнерів, самі контейнери побудовані з образів. Подібні образи використовуються в віртуальних машинах та хмарних середовищах. Образ, також можна створити з контейнера, що вже використовується наприклад локальним хостом, або віддаленим, така дія називається публікацією. При цьому, образи можна пересилати частинами або повністю, за допомогою їх хеша, якими вони ідентифікуються, також кожний образ має індивідуальні властивості та ім’я, за допомогою яких, їх можна знайти в сховищі. Образи із серверів можуть зберігатися в кеші від десяти дні і більше, зважаючи на налаштування.

LXD підтримує три сховища: ubuntu, ubuntu-daily, images

- ubuntu – сховище образів Ubuntu;

- ubuntu-daily – сховище денних збірок Ubuntu;
- images – сховище користувацьких образів. [2]

Встановлені образи оновлюються автоматично, в залежності від параметрів налаштування, то ж розробник завжди має найновішу версію.

1.4.2. Параметри безпеки LXD

Основною рушійною силою при створенні LXC а потім і LXD було намагання створити безпечні контейнери на основі Linux дистрибутивів.

- Kernel namespaces. LXD почала використовувати більш продвинутий та безпечний простір імен ніж LXC. В той час як LXC використовує непривілейовані контейнери, LXD дозволяє надати певним контейнерам привілею, якщо це необхідно для користувача, але залишаючи при цьому систему-хоста ізольованою;
- Seccomp. Для збереження системи, LXD використовує фільтри, при знаходженні системних викликів, які можуть завдати шкоди;
- CGroups. Використання ресурсів системи завжди обмежено в LXD, тому система-хост завжди захищена від несанкціонованих атак з боку контейнерів.
- AppArmor. Для ізоляції контейнерів та зменшенню їх можливої взаємодії, LXD додає покращені обмеження для сокетів, доступу до файлів та використанні mount;
- Capabilities. Як вже було сказано, контейнерна система дозволяє створювати велику кількість екземплярів, а ця функція відповідає за можливість недопускання перевантажень з боку ядра. [2]

Загалом така система допомагає користувачу абстрагуватися від постійної перебудови параметрів як це було в LXC, але при цьому залишає таку можливість, така реалізація в LXD називається «мова конфігурацій».

1.4.3. REST API

Вся взаємодія між LXD та клієнтом відбувається за допомогою REST API [2]. Це одна з головних відмінностей LXD від LXC, оскільки клієнт може взаємодіяти з демоном тільки через REST API. При наявності доступу до HTTPS, через сертифікат аутентифікації, доступ до REST API відкривається через спеціальний локальний unix-socket. Websocket дозволяє спілкування LXD при необхідності більш складної взаємодії. API завжди є актуальною при встановленні нового LXD, але при цьому в нього працює сумісність з попередніми версіями, а також можливість встановлення розширень, що покращують роботу з API.

1.4.4. Масштабованість

Хоча інструменти, які надає LXD, досить зручні і функціональні, але самі по собі вони не дозволяють керувати великою кількістю хостів, зі ще більшою кількістю контейнерів. Щоб забезпечити таку можливість, частіше за все використовують додатковий плагін nova-lxd в OpenStack [2], цей плагін дозволяє керувати контейнерами подібно системі керування віртуальними машинами. Для керування всіма складовими системи, такими як сховища, мережа та навантаження технологія OpenStack дозволяє користуватися його API, що полегшує роботу користувача.

1.5. Solaris Containers

Solaris Containers або Solaris Zones це технологія контейнеризації на основі операційної системи Solaris, що дозволяє ділити цю систему на програмному рівні, створюючи тим самим контейнери. Такі контейнери ще називаються зонами, що мають свої власні незалежні ресурси системи та користувачів [3]. Структура такої системи показана на рис.1.2.

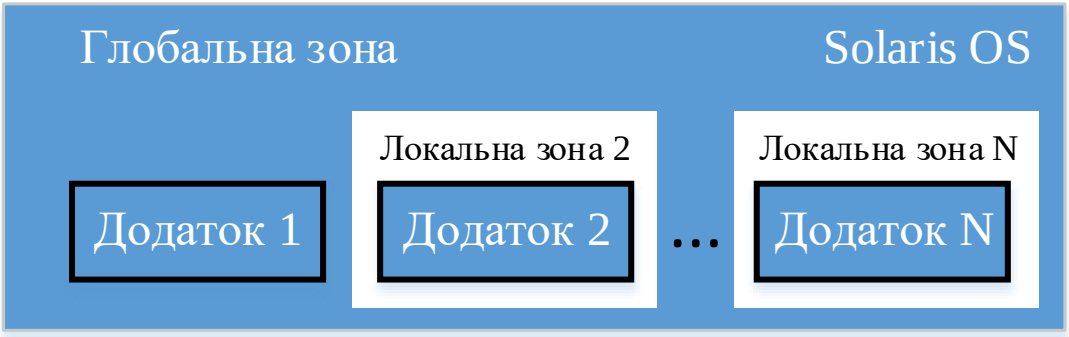


Рис. 1.2. Структура системи контейнерів Solaris Containers

Зони в системі Solaris Containers діляться на два типи, глобальна та локальна зони. Глобальною зоною називають ту операційну систему, яка є основною, тобто та, яка запускається при включенні комп'ютера і містить всі ресурси системи хоста. Локальними ж зонами називають ті самі контейнери і частіше всього вони називаються просто зонами, яким надаються лише певні ресурси системи, а при назві основної операційної системи вживають приставку «глобальна». Для забезпечення безпеки та попередження всіх ризиків, робота всіх локальних зон строго ізольована одна від іншої, за виключенням головної зони. Глобальна зона завжди так чи інакше має доступ до всіх локальних зон та процесів що в них відбуваються, а от самі зони не мають ніякого впливу на головну. Отже в системі Solaris Containers завжди присутня хоча б одна зона – глобальна, а інші зони можуть визначатися за потребою [3].

1.5.1. Плюси системи Solaris Containers

Загальне навантаження зон на систему досить мале, операційна система може підтримувати кілька тисяч локальних зон без суттєвого навантаження. Кожна зона містить свій власний ID, що ідентифікує її серед інших, мережевий інтерфейс та власну систему зберігання даних. Локальні зони можуть зберігати як програмне забезпечення із основної системи, так і додаткові продукти, що не встановлені в глобальній зоні. Зони містять списки користувачів, який може бути складений індивідуально, але при

цьому система підтримує використання однакових ідентифікаторів користувачів для різних зон.

Зручні налаштування, допомагають легко керувати ресурсами через глобальну зону [4]. Зоні може бути надані певні ресурси системи, пам’яті, процесорів та іншого щоб забезпечити мінімальні витрати. Кожна зона може знаходитись у певному стані:

- Налаштовано: закінчення конфігурації, його підтвердження;
- Незавершено: стан переходу між встановленням та видаленням;
- Встановлено: стан при успішному встановленні пакетів;
- Запуск: стан після успішного завантаження зони;
- Готово: така зона повністю доступна для використання;
- Завершення роботи: стан зупинення, в якому зона залишається перед повним виключенням;
- Завершено: зона зупинена [3].

Досить великим плюсом є використання Branded zones, що дає можливість Solaris Containers, не зважаючи на використання одного ядра, виконувати в локальних зонах екземпляри операційних систем, відмінних від тієї що використовує глобальна зона.

1.5.2. Мінуси системи Solaris Containers

Основним мінусом Solaris є те, що вона працює тільки з однією системою, тобто вона має досить малий потенціал для масштабування і роботи з хмарними структурами. Також потенційним мінусом є те, що деякі програми, загалом ті що використовують функції ядра, або ті які потребують певних привілеїв, не можуть бути виконаними у локальній зоні, так як вона не має власного ядра.

Контейнери Solaris є досить непрактичними для виконання сценаріїв безперервної доставки, що зумовлено неможливістю швидкого запуску контейнерів на основі образів.

Також Solaris Containers не може конкурувати з контейнерами Linux у плані керування, оскільки на відміну від десятків варіантів інструментарію для керування та моніторингу контейнерів в Linux системах, Solaris має тільки ті рішення, які надає Oracle [4].

1.6. Rkt (Rocket)

CoreOS, як окремий проект розробив свій інструмент для керування контейнерами Rocket пізніше перейменований в скорочення rkt. Переслідуючи ідею створення ізольованої контейнерної системи CoreOS намагалася створити альтернативу контейнерам Docker, зробити свою систему більш підходящою для серверів та з кращою адаптацією під інші системи, також стверджувалося, що rkt повинна бути більш безпечною за Docker через застосування нової системи виконання. Також CoreOS розробив спеціальні характеристики App Container, на ї основі була побудована система маніпуляції контейнерами, яка основною ціллю обирає створення формату контейнеру з універсальною портативністю [5].

Головню перевагою rkt являється спеціалізована архітектура, яка на противагу Docker, де один процес у фоновому режимі централізовано керує всіма операціями, використовує модульну систему, що є по суті багаторівневою архітектурою. Такий підхід дає переваги в забезпеченні безпеки та масштабованості, оскільки робота з контейнерами ділиться на певні шари або етапи, на кожному етапі виконується певна дія, від початку налаштувань, до запуску контейнера та програми [5]. За рахунок можливості підключення додаткового інструментарію, можна розширити функціонал тим самим збільшити варіативність реалізації технології.

1.6.1. Стадії запуску контейнера

Як вже було сказано, rkt має багаторівневу архітектуру, побудовану на модульній системі, що складається зі спеціальних етапів або, як їх ще

називають стадії. Кожна стадія відповідає за свої певні налаштування та дії. Загалом існує три стадії запуску rkt контейнера, їх назви відповідають ітераціям, тобто послідовності дій:

0. Початкова, або нульова стадія. Відбувається підготовка контейнера, тут rkt не використовує допоміжні ресурси. Створюється UUID ідентифікатор та вказівки на його основі, далі на основі потреб контейнера генерується файлова система з налаштованими під нього директоріями. Відбуваються налаштування для запуску наступних стадій, а саме в файлову систему контейнера копіюються виконуючі файли для використання в наступній стадії та копіювання програм з розпакованого образу в створені директорії для кінцевої стадії.
1. Перша стадія. Виконується генерація виконуваної групи яка ґрунтується на файловій системі, що була попередньо створена на попередній стадії. Виконується встановлення основних компонентів таких як іменний простір, cgroups та точок монтування. За виконання цієї стадії відповідає спеціальний виконуючий файл, який має права root користувача на виконання певних операцій, а також може керувати процесами та налаштовувати cgroup. Такий файл має назву unit system, саме через його генерацію відбувається налаштування, а для роботи оточення використовується system-nspawn.
2. Друга стадія. На цій стадії виконується активація програми яка була закладена в контейнер [5].

1.6.2. Особливості rkt

- Можливість виконання зміни конфігурації для будь якої програми;
- Можливість інтеграції з контейнерами Docker, при умові їх побудові на моделі App Container;
- Не використовує центральний демон, для модулів використовується окремі середовища, що є ізольованими один від одного;

- Використання специфікації App Container, як основної для реалізації формату контейнера, при цьому підтримуючи інші образи;
- Підтримка більшості дистрибутивів Linux;
- Посилення надійності системи за рахунок використання спеціального гіпервізора KVM для ізоляції процесів, та підтримки технологій посиленого контролю доступу SELinux та SVirt.
- Вилучення використання методів, при яких використовуються поділ привілеїв, для програм які використовують права доступу root, таким чином відходячи від використання єдиного керуючого процесу. Всі дії дозволяються непривілейованому користувачу на відміну від Docker з необхідністю прав root.
- Використання ізольованого оточення для запуску контейнерів, замість централізованого фонового процесу [6].

Висновки до розділу 1

Наразі існує багато технологій для запуску програми або коду на сервері, при цьому головною проблемою є безпека серверної системи при запуску таких програм. Головним рішенням цієї проблеми є віртуалізація системи, а точніше конкретна контейнеризація, що надає ізольованість і можливість виконувати будь які програмні продукти, без небезпеки для основної системи.

Ринок контейнерних технологій є досить великим, більшість світових компаній так чи інакше використовують та розвивають ці технології. Розглянуті в цьому розділі системи контейнеризації можуть використовуватися для поставлених цілей проекту, але їх підхід є досить специфічним, та має значні обмеження, що робить необхідним обрання іншої моделі контейнеризації, а саме технології Docker.

РОЗДІЛ 2

ТЕХНОЛОГІЯ DOCKER

Docker являє собою платформу для розробки, публікації та запуску додатків [7]. Docker надає змогу розділити додатки від інфраструктури і надає можливість швидкого розгортання. За допомогою цього з'являється можливість керування інфраструктурою з такою легкістю як із додатками. Для спрощення процесу розгортання проекту на сервері можна використовувати docker для його завантаження, розгортання та тестування, який робить це за допомоги контейнерної віртуалізації з використанням процесів та утиліт, що допомагають керувати програмами та їх розгортати.

Важко уявити додаток який не підтримується ядром docker'у для його безпечного запуску в ізольованому контейнері. Технологія безпечної ізоляції docker'у надає змогу одночасному запуску багатьох контейнерів на одному хості. Проста та доступна реалізація контейнера не створює додаткового навантаження на гіпервізор, що надає змогу отримати більшу обчислювальну потужність від сервера.

2.1. Архітектура Docker'а

Docker представляє собою програмне забезпечення, яке складається з:

- Платформи з відкритим кодом для віртуалізації;
- PaaS для керування і створення контейнерів.

Для цього розробниками платформи docker'а було прийнято рішення використання клієнт-серверної архітектури, тобто docker-клієнт обмінюється інформацією з демоном docker'у, який у свою чергу бере на себе відповідальність за створення, запуск та розміщення контейнерів на сервері. Docker надає гнучку систему налаштування процесу спілкування сервера та клієнта. Спілкування може відбуватися на одній операційній системі або ж є можливість віддаленого підключення до демона Docker'у. Також можна

обрати спосіб спілкування клієнта та сервера за допомогою сокета чи RESTful API.

На діаграмі (рис.2.1) продемонстровано запуск демону на обраній хост-машині. Безпосередньої взаємодії з сервером не відбувається, для організації спілкування використовується клієнт. Docker-клієнт - головний інтерфейс системи, що використовується для отримання команд від користувача та взаємодії з docker демоном.

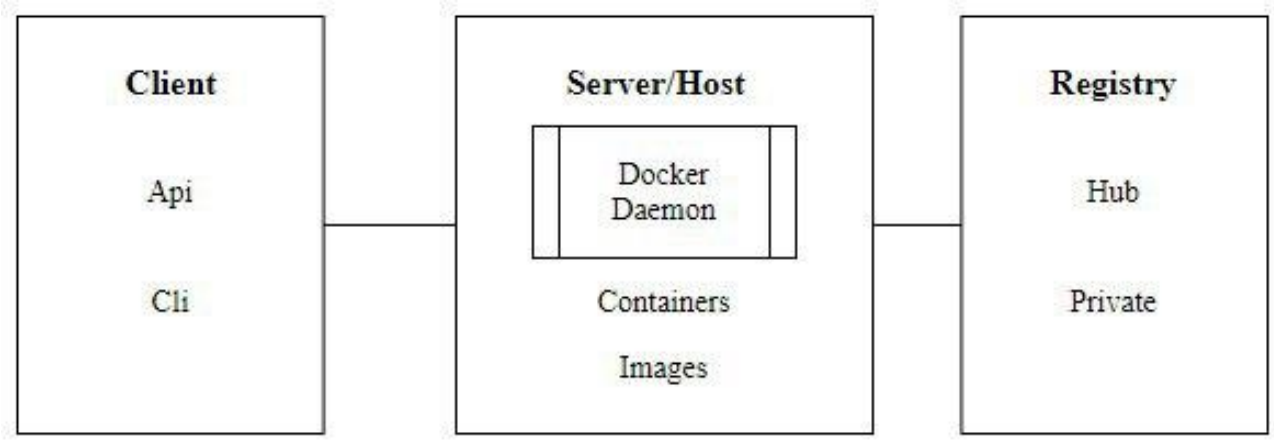


Рис.2.1. Діаграма запуску демону на хост-машині

Шаблон docker образу працює за принципом read-only. Для прикладу, образ може бути створений з використанням Linux з будь-яким http-сервером і додатком. Контейнери створюються за допомогою образів. Docker надає можливість маніпулювати образами, а саме:

- створювати нові;
- оновлювати існуючі;
- завантажити образи, створені іншими людьми.

2.2. Головні складові компоненти Docker

Docker складається з трьох основних компонентів, а саме:

1. Images (образи). Образ являється основою для контейнеру. Вони поділяються на базові та дочірні.

Базові образи не мають батьківського образу, переважно це образи операційних систем, а саме linux дистрибутиви, busybox, windows.

Дочірні образи є базовими, але додатково сконфігуровані і мають додаткову функціональність.

Існують офіціальні та користувацькі образи, кожен з яких може бути базовим чи дочірнім.

Офіціальний образ підтримується командою розробників Docker. Користувацькі образи створюють користувачі у складі яких лежать базові.

2. Registries (реєстри). Реєстр потрібен для зберігання образів, він може бути публічним або приватним. З реєстрів можна завантажити вже існуючий образ або додати свій.
3. Containers (контейнери). Контейнер – це те саме, що директорія. Всередині нього знаходяться всі файли, що потрібні для запуску додатку. Всі контейнери створюються з образу, їх можна створювати, запускати та зупиняти. Перевага контейнерів в тому, що вони є ізольованими та безпечними для додатку. [7]

2.3. Принцип роботи Docker

Образ – шаблон, з якого створюються контейнери. Він складається з декількох рівнів, використовує UnionFS, яка використовується для поєднання рівнів в один обрах. Це дозволяє створювати когерентну файлову систему для накладання директорій з різних файлових систем [7].

Саме використання такої системи робить docker легким. Якщо відбувається зміна образу, через оновлення додатку, створюється новий рівен. Таким чином, можна уникнути заміни всього образу та запуск нової збірки, на відмінну того, що відбулося при використанні віртуальних машин, а докер тільки додає та оновлює рівні. Це дозволяє не поширювати повністю новий образ, а публікувати тільки оновлення та забезпечує швидку та просту доставку оновлень.

Всі нові образи створюються з базових образів. Вони створюються за допомогою використання певних інструкцій, які містять такі дії:

- Запуск команди;
- Додавання файлу чи директорії;
- Створення змінних оточення;
- Задавати порядок запуску дій після створення контейнеру.

Всі інструкції розміщуються у спеціальному файлі, який називається Dockerfile. Docker налаштований на зчитування цього файлу при створенні образу, він виконує команди, описані у цьому файлі, та повертає готовий образ [8].

Реєстр є сховищем для образів. Після створення образи ви можете поширити його, використовуючи публічний реєстр Docker Hub, або розмістити на вашому особистому приватному реєстрі. Використовуючи клієнт docker'у, можна знайти і завантажити вже опубліковані образи, і завантажити на свій сервер для налаштування контейнерів.

Docker Hub надає доступ всім до публічних сховищ образів. Якщо ви не маєте доступу до приватних образів, то ви не побачите їх у результатах пошуку. Тільки якщо це ваш образ чи ви є його користувачем, можете отримати доступ до нього і створити контейнер [8].

2.4. Принцип роботи контейнера

Docker контейнер складається з операційної системи, яка призначена для файлів і метаданих користувача. Образ надає інформацію контейнеру, що саме в ньому повинно знаходитися, які інструкції запустити, момент його запуску та інші інструкції. Ви маєте права тільки на читання образу, після його створення. Коли docker починає запускати контейнер, то створює рівень читання-запису поверх образу (використовую union file system), в якому вже може бути відкритий ваш додаток [8].

Використовуючи програму docker чи RESTful API, docker клієнт дає команду демону на запуск контейнеру.

На рис. 2.2. описується інструкція для запуску docker клієнта

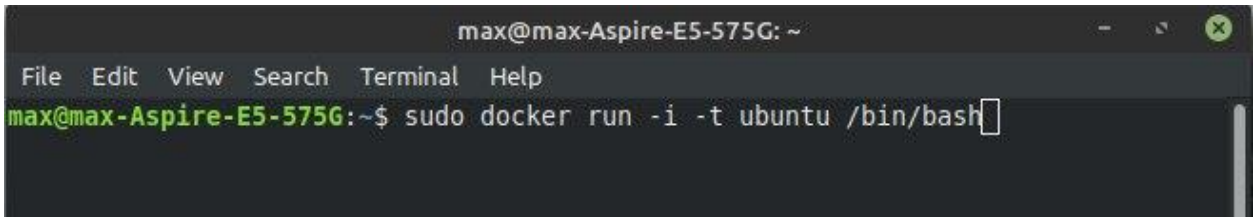


Рис.2.2. Запуск docker клієнта

Опція run вказує на те, що буде відкритий новий контейнер. Для запуску контейнеру є мінімальні вимоги, а саме такі атрибути:

- Образ, який ви використовуєте для створення контейнера. У прикладі це Ubuntu;
- Команда, яка буде запущена після створення контейнеру /bin/bash.

Детально про те що відбувається у момент запуску даної команди:

- Перевірка наявності образу ubuntu спочатку на машині користувача, у випадку, якщо образу немає, відбувається пошук і завантаження образу з Docker Hub. Якщо образ існує, то docker використовує його для створення контейнера;
- Коли образ знайдений та завантажений, docker починає створювати контейнер;
- Створення та ініціалізація файлової системи. Монтування рівня (доданого образу) у режим read-only;
- Створення мережевого інтерфейсу та його ініціалізації, для спілкування docker'у з хост машиною;
- Встановлення IP адреси;
- Запуск додатку;
- Оброблення та вивід користовачу вихід на додаток, а саме підключення стандартних входу, виходу та потоку помилок додатку для відслідковування процесу роботу програми.

Після цієї команди ви вже отримуєте робочий контейнер. На даному етапі ви можете взаємодіяти зі своїм додатком. Якщо ви вирішили його зупинити, буде достатньо просто видалити контейнер.

2.5. Технології, використання при розробці Docker

Docker був розроблений з використання мови програмування Go та деякими можливостями ядра Linux, для реалізації наведеного вище функціоналу.

У docker використана технологія namespaces, щоб організувати ізольовані робочі простори для контейнерів. У момент запуску контейнера, docker створює певний набір просторів імен для нього. Це необхідно для створення ізольованого рівня, кожен з ваших контейнерів має доступ до свого простору імен, і не може отримати доступ до операційної системи користувача.

Основні простори імен, задіяні docker'ом наведені нижче:

- Pid – потрібен для ізоляції процесу контейнера;
- Net – для налаштування мережі;
- Ipc – управління IPC (InterProccess Communication) ресурсами;
- Mnt – потрібен для роботи точок монтування;
- Utc – створює ізоляцію ядра і забезпечує контроль генерації версій.

Docker використовує технологію cgroups (Control groups) – це потрібно для надання додатку тільки тих ресурсів, що він потребує. Ця технологія забезпечує ще один рівень ізоляції і гарантує, що контейнери будуть гарно працювати один з одним. Також контрольні групи допомагають встановити обмеження контейнерам на ресурси користувача. Як приклад, обмежити максимально можливу кількість пам'яті, яку використовує контейнер.

Принцип роботи UnionFS полягає у створенні рівнів, що забезпечує її легкість та швидкодію. UnionFS допомагає Docker'у створювати блоки, які є

необхідністю для створення контейнерів. AUFS, btrfs, vfs, DeviceMapper – основні варіанти UnionFS, задіяні Docker'ом [9].

Все це Docker поєднує у так звану обгортку, що є форматом контейнеру. Libcontainer – формат бібліотек docker'а за замовчуванням. Також за допомогою технології LXC ви можете користуватися традиційним форматом контейнерів Linux систем.

2.6. Використання журналів у Docker

Docker підтримує технологію запису інформації про контейнери до журналу. Для цього існують команди:

- Docker logs – виводить всю інформацію, яка була записана до журналу запущеним контейнером.
- Docker service logs – відображає інформацію, яка була записана усіма контейнерами, які беруть участь у сервісі.

Результат виводу та формат роботи цих команд залежить від того, як була налаштована кінцева точка команди контейнеру.

Docker logs та docker service logs – мають такий же формат виводу як і запуск цих команд безпосередньо у терміналі, за замовчуванням. Тобто запис буде відбуватися в три основних потоки STDIN, STDOUT, та STDERR як це відбувається в Unix чи Linux подібних операційних системах [10].

- STDIN – потік для введення команд, який може включати ввід інформації з клавіатури чи з іншої команди.
- STDOUT – потік для виводу результату команд.
- STDERR – потік, який використовується для виводу помилок, які можуть виникати при роботі програм.

Також всі ці потоки можна перенаправляти у разі потреби, наприклад, до файлу чи бази даних.

Журнал може бути менш інформативний, якщо використовувати такі сценарії використання:

- Встановлення logging drivers, які необхідні для запису журналу у файл, іншого сервера, бази даних.
- Образ, який запускає веб-сервер чи базу даних, ці програми можуть надсилати свої журнали в інші місця замість стандартних потоків docker'у.

При використанні драйверів для журналювання, краще не використовувти команди docker logs та docker service logs. Для цього треба використовувти сховище, яке було налаштоване у драйверах, для перегляду журналів.

Якщо ви запускаєте веб-сервер чи базу даних docker має офіційні образи nginx та httpd, які можуть допомогти вирішити проблему.

Nginx створює посилання з /var/log/nginx/access.log до /dev/stdout та з /var/log/nginx/error.log до /dev/stderr, перезаписує журнал та спричинює переміщення журналу до специфічного девайсу [10].

Httpd драйвер змінює конфігурацію додатку на запис журналів до /proc/self/fd/1, що є STDOUT та помилок до /proc/self/fd/2, що є STDERR [10].

2.7. Використання Dockerfile для створення образів

Для створення образів Docker доцільно використовувати спеціальний файл Dockerfile. Даний файл являється інструкцією з набором упорядкованих команд, порядок яких дає можливість створенню необхідного образу за рахунок керування терміналом системи [11]. Dockerfile надає команди в певному стандартизованому форматі та певним набором початкових інструкцій. Загалом, як відомо образи Docker являють собою певний набір рівнів [11], які допомагають використовувати один образ для багатьох завдань, інструкції в Dockerfile можна розглядати, як окремі рівні, оскільки інструкції певним чином змінюють стан образу, тоді кожна наступна інструкція є певною зміною попередньої.

2.7.1. Інструкції Dockerfile

2.7.1.1. Інструкція FROM

Дана інструкція являється початковою для будь-якого Dockerfile, тому вона має стояти на першому місці для ініціалізації початку зборки. ЇЇ виклик створює перший рівень зборки з початковим образом. Початковий образ встановлюється користувачем та може бути як власним так і завантаженим з будь якого доступного репозиторію. Один Dockerfile може містити кілька викликів інструкції FROM, в такому випадку може бути створено стільки образів, скільки разів дана інструкція викликалась, або вони можуть створювати залежності між собою. В разі використання додатку AS name до інструкції, користувач може вказати назву для цього рівня, яка буде являтися ідентифікатором та яку в подальшому можна буде використовувати в наступних інструкціях даного Dockerfile [12].

Інструкція FROM має два необов'язкових параметри, це значення тегу та платформи. Якщо назва тегу не буде зазначеною, тоді система буде використовувати за замовчуванням тег latest, при створенні нового образу, платформа ж встановлюється через використання флагу --platform [12] та вказуючи необхідну платформу для образу, якщо ж його не використовувати тоді платформою образа буде обрана та, яка є цільовою для запиту.

2.7.1.2. Інструкція WORKDIR

Головним завданням даної інструкції є вказівка на шлях до директорії, яка буде вважатися робочою. На цей шлях будуть посилатися всі наступні інструкції, як на шлях до робочого каталогу. Якщо WORKDIR викликається кілька разів, то з кожним новим разом шлях доповнюється новою ланкою, записаною в інструкції, а кожний наступний шлях буде знаходитися відносно попереднього. За замовчуванням робочою директорією являється кореневий

каталог, тому якщо не вказувати ніякого шляху, WORKDIR буде створений самостійно в ньому.

Дана інструкція в якості параметру може використовувати змінні оточення, які були оголошені перед викликом WORKDIR в цьому ж файлі за допомогою ENV. [12]

2.7.1.3. Інструкція COPY

Ця інструкція являє собою передатчик даних до файлової системи контейнера. Вона приймає два параметри, перший src вказує на директорію або файл який необхідно скопіювати, другий dest – шлях до місця в контейнері, куди ці дані будуть збережені. У параметр src можна вказувати декілька джерел, при цьому за замовчуванням їх пошук буде відбуватися у контексті зборки, якщо ж джерела розташовані у каталогах які вкладені в контекст, шлях необхідно прописувати повністю. Шлях dest за замовчуванням вказує на кореневий каталог робочої директорії у контейнері, але дає можливість змінювати шлях, на необхідний користувачу. [12]

Важливою особливістю є те, що при першому виклику інструкції COPY, якщо src була змінена, весь кеш стає недійсним для інструкцій, які йдуть після неї.

2.7.1.4. Інструкція RUN

Дана інструкція використовується для виконання певних команд, для створення нових шарів образу, після чого фіксує результати. В наступних Dockerfile інструкціях буде використовуватися цей зафіксований образ, таким чином фіксації не є ресурсномісткими, а контейнери можуть бути створеними на будь якому етапі розвитку образу, відповідно до понять, які закладені в Docker.

Інструкція RUN може бути написана в двох формах: shell та exec. Форма shell використовується інструкцією за замовчуванням та використовує

стандартну для Linux оболонку /bin/sh -c, щоб запустити іншу оболонку у формі shell можна скористатися командою SHELL. Також, для перенесення команди RUN на наступний рядок в формі shell використовується символ зворотного слешу - \. Форма exec дає можливість використовувати команди без оболонки, тобто їх стандартний вид, тим самим уникаючи обробки оболонки та зміни команди на їх основі.

Також при використанні форми exec є можливість надавати оболонку безпосередньо. При цьому при використанні оболонки, саме вона робить розширення середовища, а не docker. [12]

Доцільним є те, що кеш інструкції не знищується при переході на наступну збірку. Збережений кеш може бути необхідний при наступних зборках, таким чином його можна буде використовувати повторно. Оскільки автоматичне очищення кешу не виконується, команда docker build має спеціальний флаг --no-cache, використання якого видаляє весь кеш [12]. Також він може бути видаленим інструкціями ADD або COPY.

2.7.1.5. Інструкція CMD

Головною ціллю даної інструкції є вказування на точку входу процесу у контейнері, тобто її параметром вказується виконуваний файл додатку який його запускає, або інструкція ENTRYPOINT, якщо такого файлу немає. Внесення більше однієї інструкції CMD не принесе користі, оскільки виконується тільки остання з перелічених. При використанні ENTRYPOINT обидві інструкції вказуються, як масив JSON [12]. Поєднання цих інструкцій надає можливість кожного разу запускати один і той же файл в контейнері.

Також інструкція має дві форми shell та exec, використання shell надає оболонку команді, а exec – ні, при цьому використання другої форми вважається кращим вибором.

2.8. Команди які використовуються при роботі з Docker

2.8.1. docker build

Для того щоб автоматизувати створення образів використовується команда docker build, яка замінює собою написання кількох команд в терміналі командного рядка, та впорядковано виконує їх дії. Дана команда створює образ використовуючи та обробляючи Dockerfile та всі файли, які знаходяться в сховищі або каталозі, ці файли називаються контекстом і на них вказує PATH чи URL. Обробка файлів відбувається рекурсивно, тобто вказуючи шлях до каталогу, команда почне обробляти не тільки ті файли які зазначені за даним шляхом, але і всі інші які розміщуються у вкладених каталогах. Тому переважно краще використовувати нові, або просто пусті каталоги та переносити туди Dockerfile та тільки необхідні файли, адже чим більше файлів оброблюватиме демон Docker тим більше буде завантажена система.

Для зменшення навантаження доцільним є використання файлу .dockerignor який додається в корінь контексту, і містить інструкції для вилучення з обробки контексту певних файлів та каталогів. Оскільки першочергово при обробці каталогів шукається саме .dockerignor, то при його наявності в корені система проігнорує всі вказані в ньому файли що покращить працездатність всієї системи. За замовчуванням контекстом є той каталог в якому знаходиться Dockerfile до якого був написаний шлях, щоб змінити шлях на інший використовується флаг -f в команді docker build, після якого вказується новий PATH до необхідного файлу Dockerfile. Також є можливість зберегти отриманий образ після його зборки використовуючи флаг –t та шлях до сховища, при цьому зберігати можна одразу в кілька сховищ, перелічуючи їх при написанні команди docker build [12].

Як вже було сказано, інструкції з Dockerfile виконуються демоном Docker по черзі, то ж результати виконання кожної з них зберігаються на

деякий час в кеш образів, тобто по суті кожна інструкція створює новий образ, при чому інструкції виконуються незалежно одна від іншої. Така система дає сильне пришвидшення для роботи docker build при виконанні однакових запитів, оскільки при необхідності потрібні образи не створюються заново а просто беруться з кеша. Обмеженням такої системи є використання образів з локального батьківського ланцюга, тобто були створені попередньо, або якщо весь ланцюг був завантажений через docker load. Використовуючи опцію --cache-from, можна взяти відповідний образ з кешу та використати його, при цьому вони можуть бути сторонніми та для них не є обов'язковою наявність батьківського ланцюга. [12]

2.8.2. docker run

Створення або скачування образів docker, створення файлу конфігурацій і все інше відбувається для запуску на сервері контейнера, це являється основною причиною використовувати технологію Docker – запускати програми, операційні системи та інші процеси не в основній системі сервера, а у окремих контейнерах, які розташовані в ній, але при цьому не мають на неї впливу. Команда docker run відповідає за запуск контейнера та його параметри.

Першим параметром даної команди є ідентифікатор відповідного образу docker, на основі якого буде запущений контейнер. Основною силою даної команди є використання опцій, а саме те, що через них можна змінювати налаштування, які були встановлені на образ при його розробці і навіть змінювати більшість параметрів, які були визначені Docker-ом за замовчуванням. Можливість змінювати параметри за замовчуванням робить цю команду найскладнішою з точки зору налаштувань, адже вона має найбільшу кількість параметрів серед всіх інших команд docker. Перший параметр відповідає за режим в якому буде запущений контейнер. Існує два

режими запуску контейнера: фоновий «окремий» та режим за замовчуванням «переднього плану» [13]:

Окремий режим – в цьому режимі контейнер перестає керуватися командами з терміналу, щоб передати або прийняти інформацію з нього необхідно створити окреме підключення або використати спільний простір. Для встановлення цього режиму використовується параметр -d в купі з параметром --rm. Якщо параметр --rm не буде встановлений, тоді контейнер закриється при завершенні процесу, який він містить, якщо ж даний параметр встановлений, тоді такий контейнер буде видалений при його зупиненці , або при зупинці демона, в залежності від того що трапиться раніше.

Режим попереднього плану – цей режим є стандартним, для його запуску достатньо не використовувати параметр –d, а отже він є протилежністю до фонового режиму [13]. При такому режимі консоль має доступ до керування стандартними вводом та виводом з контейнера, та виводом помилок, та починає імітувати термінал.

Як вже було сказано, параметри даної команди дають можливість змінювати стандартні налаштування контейнерів, а це означає, що виникає можливість обмежування ресурсів наданих системою окремим контейнерам. Обмеження в ресурсах не встановлюються за замовченням при створенні контейнеру, а отже немає ніякого контролю за ресурсами системи, кожний контейнер бере все що надає йому планувальник. В такій ситуації при запуску великої кількості контейнерів система може не витримати та видати збій. Так ядро Linux при нехватці пам’яті може почати видаляти другорядні процеси, для надання отриманої пам’яті системі, а оскільки процеси зв’язані з Docker другорядними то вимкнутися можуть і вони, що є недопустимим [14]. Наразі шанс видалення демону менший ніж шанс видалення контейнеру, але він все ж таки існує. Тому доцільним є використання додаткових обмежуючих параметрів.

Для обмеження пам’яті яку надає система використовується параметр -m після якого вказується кількість пам’яті виділеної для контейнера, якщо її не вказувати, то виділена пам’ять буде обмежена чортма мегабайтами, також існують модифікації даного параметра:

-m-swap – цей модифікатор дає можливість контейнеру використовувати диск замість пам’яті, якої йому не вистачає;

-m-swappines – впливає на обмін анонімними сторінками, може приймати значення 0, якщо необхідно закрити обмін та до 100 якщо необхідно відмітити можливість їх заміни;

-m-reservation – є полегшеною версією параметру -m, він не вказує на строгу заборону у використанні пам’яті і не забезпечую гарантій;

--kernel-m – модифікатор, який обмежує контейнеру використання пам’яті ядра, при цьому необхідно розуміти, що в будь якому разі мінімальний об’єм повинен використовуватися. [14]

2.8.3. docker container stop

Для того, щоб регулювати роботу контейнера та час затрачений на використання контейнером ресурсів системи, можна встановлювати таймер, який починає працювати під час запуску контейнера. Коли час вичерпується тоді використовується команда docker container stop, яка являє собою примусову зупинку контейнера або контейнерів всередині яких ще не закінчилось виконання процесів. Спочатку відбувається очікування зупинки процесу всередині контейнера, цей час є резервним, на випадок, якщо процесу не вистачило невеликої кількості часу, після чого він знищиться [15]. Також за допомогою параметра -t можна надати часу очікування інше значення, за замовчуванням воно дорівнює десять секунд.

2.8.4. docker rmi

При закінченні роботи контейнера, та якщо його образ більше не буде використаний, чекати нового запиту для його використання не вигідно, тому є доцільним видалення цього образу із працюючих систем, для звільнення ресурсів системи для інших процесів. Команда docker rmi видаляє один або кілька образів з процесів системи [16]. Дана команда не може видалити образи які знаходяться в реєстрі, а також за замовчуванням при працюючому контейнері, його образ не може бути видаленим, але при використанні в команді параметру -f це налаштування зміниться, та образ буде видалений примусово.

Щоб видалити образ достатньо передати в команду його ідентифікатор або тег, якщо він один, якщо тегів у образа декілька то видалиться тільки вказаний тег. Також, якщо необхідно видалити кілька образів з однаковими ідентифікаторами в різних сховищах, можна використати вище вказаний параметр -f, тоді видаляться всі відповідні цьому ідентифікатору образи і теги. [16]

2.8.5. docker images

Дана команда видає список всіх образів, які в даний момент наявні на верхньому рівні, а також надає їх параметри: сховища в яких вони знаходяться, їх теги, ідентифікатори, розмір та приблизний час створення [17]. Для знаходження всіх образів на сервері використовується команда docker image ls. Також надається можливість виводу конкретної інформації по образам,застосовуючи відповідні параметри в команді. Також, якщо використовувати в команді ідентифікатор або тег образу, у відповідь будуть виводитися тільки ті образи які відповідають даним параметрам.

Висновки до розділу 2

Для розробки системи завантаження та запуску коду на сервері була обрана саме технологія контейнеризації Docker.

Порівнюючи з іншими подібними технологіями, Docker надає більш легкі для розуміння та більш мінімалістичні контейнери. Це робить дану технологію більш підходящою для розгортання не повноцінних операційних систем, а лише додатків для запуску коду певної мови. Такий підхід зменшує навантаження системи, при цьому надаючи найбільш безпечні умови для перевірки програмного коду. Вільна можливість в користуванні системою також вплинула на цей вибір, оскільки система розробляється з найменшими витратами.

Використання Dockerfile та розвинута система команд надають широких можливостей для контролю за системою та розподілом ресурсів. А розвинена документація надає повний набір необхідної інформації для використання технології.

РОЗДІЛ 3

СТРУКТУРА СИСТЕМИ

Розроблювана система являє собою структурований комплекс з певних модулів, класів та відповідних їм методів, відповідна діаграма класів наведена в додатку В. Система була побудована за допомогою фреймворку ASP .NET Core. Як можна побачити на діаграмі, структурно система складається з трьох класів CodeController, CodeService та CodeModel, інтерфейсу – ICodeService та додаткового enum типу LanguageType. Спільна їх робота надає можливість працювати всій системі в цілому.

3.1. Технологія ASP .NET Core

ASP .NET Core – новий кросплатформений фреймворк від компанії Microsoft для розробки веб додатків, який був розроблений на зміну ASP .NET MVC.

Перевагами цього фреймворка від ASP .NET MVC є відкритий вихідний код, що дозволяє розробникам з усього світу вносити свої зміни на платформі GitHub. Також він має готовий функціонал для забезпечення впровадження залежностей. Ще однією з переваг є відсутність залежності до IIS. Через компіляцію вихідного кода в IL-код, який є спільним для багатьох мов, а саме: C#, C++, Visual Basic, F#, їх легко об’єднувати між собою. [18]

Перша версія нової платформи вийшла у 2016 році і вже за 4 роки існування потрапила до верхівок рейтингу фреймворків за продуктивністю. Також однією з переваг є те, що мову C# розробляє Microsoft, тому знаючи лише одну мову можна займатися усіма сферами, від настільних додатків до програм з машинним навчанням.

Зараз на ринку дуже популярні хмарні сервіси. З ростом тенденції Microsoft розробила хмарний додаток Azure, який має змогу до розгортання додатків на .NET.

.NET має можливість для розгортання додатків, використовуючи платформу Docker. [18]

3.2. Клас CodeController

CodeController – клас, необхідний для налаштування маршрутизації і забезпечення приймання запитів від користувача (рис. 3.1). Даний клас є безпосередньо ланкою, яка пов'язує запити користувача з роботою сервера.

```
[Authorize]
[ApiController]
[Route("codes")]
1 reference
public class CodeController : ControllerBase
{
    private readonly ICodeService _codeService;

    0 references
    public CodeController(ICodeService codeService)
    {
        _codeService = codeService;
    }

    [HttpPost("run")]
    0 references
    public IActionResult Run(CodeModel code)
    {
        return Ok(new { Result = _codeService.Run(code.Command, code.LanguageId) });
    }
}
```

Рис. 3.1. Вміст класу CodeController

Даний клас приймає інтерфейс ICodeService, що містить метод Run та забезпечує виконання коду. ICodeService ініціалізується за допомогою використання dependency injection. Метод контролеру Run приймає тип CodeModel, який являється описом об'єкту коду, що містить поля Command і LanguageId. Парамерт Command містить код, який користувач хоче запустити на сервері, LanguageId – обрана мова програмування на якій цей код написаний. Цей метод повертає результат у форматі Json з полем Result в якому міститься результат виконання програми.

3.3. Клас CodeService

CodeService – клас, що відповідає за запуск коду на сервері, містить основні методи для роботи системи, та пов’язує з нею технологію docker. Він складається з методів:

- Run
- CreateDockerFile
- CreateCommandFile
- RunDockerImageBuildScript
- RunDockerContainer
- RemoveDockerImage
- GetProcess

Також був створений enum LanguageType, що складається з трьох значень:

- CSharp
- Java
- Python

При розробленні цієї структури був використаний шаблон Facade, тобто на вихід цей сервіс має лише один публічний метод Run. Це гарантує правильний порядок виконання коду та полегшує завдання, якщо цей код використовуватимуть інші люди.

CreateCommandFile (рис.3.2) – це метод, який забезпечує створення директорії в якій відбуватимуться наступні дії та створення в ній файлу з кодом який надійшов у запиті. Приймає параметри command та languageId. Параметр command – код, який надсилає користувач, languageId – обрана мова, на основі якої буде перевірятися код. Спочатку надається випадкове ім'я для назви директорії та імені файлу з кодом. В залежності від обраної мови створюються різні шляхи до коду та обирається правильне розширення файлів відповідно до обраної мови програмування. Результат цього методу є

об’єкт, що складається з directoryPath – відповідного маршруту до створеної директорії та filename.

```
1 reference
private (string directoryPath, string fileName) CreateCommandFile(string command, LanguageType languageId)
{
    var directoryName = Guid.NewGuid().ToString();
    var directoryPath = Directory.CreateDirectory($"{Environment.CurrentDirectory}/{directoryName}");
    var fileName = Guid.NewGuid().ToString();
    var filePath = languageId switch
    {
        LanguageType.CSharp => $"{directoryPath}/{fileName}.cs",
        LanguageType.Java => $"{directoryPath}/Main.java",
        _ => $"{directoryPath}/{fileName}"
    };
    using var fileStream = File.Create(filePath);
    var fileBytes = new UTF8Encoding(true).GetBytes(command);
    fileStream.Write(fileBytes, 0, fileBytes.Length);
    return (directoryPath.ToString(), fileName);
}
```

Рис. 3.2. Вміст методу CreateCommandFile

CreateDockerFile (рис. 3.3) – метод, який відповідає за створення dockerfile. Даний метод в якості параметрів приймає directoryPath, filename, languageId. Параметр directoryPath – шлях до директорії з якого буде відбуватися створення образу, fileName – ім’я файлу з кодом, languageId – ідентифікатор мови програмування, який необхідний для створення вірного dockerfile’у. Його запис відбувається у раніше створену директорію зї шляхом directoryPath та іменем файлу «dockerfile». Dockerfile використовує parent images мов програмування C#, Java, Python. Це забезпечує швидке створення образів через використання закешованих образів, нові образи створюються на їх основі. Така технологія дозволяє не перейматися за необхідність створення нових образів. Команда RUN створює шар образу та запускає команду для компіляції коду, CMD – команда, яка буде виконуватися після створення контейнеру.

RunDockerImageBuildScript (рис.3.4) – метод, призначений для запуску створення образу. Приймає єдиний параметр directoryPath та повертає випадково згенероване значення imageId, яке є ідентифікатором створеного образу.

```
1 reference
private void CreateDockerFile(string directoryPath, string fileName, LanguageType languageId)
{
    const string dockerFileName = "dockerfile";
    var dockerFilePath = $"{directoryPath}/{dockerFileName}";
    var dockerFile = languageId switch
    {
        LanguageType.CSharp => $@"
                FROM mono
                WORKDIR /home
                COPY {fileName}.cs  .
                RUN mcs /home/{fileName}.cs
                CMD [""mono"", ""/home/{fileName}.exe""]",
        LanguageType.Java => $@"
                FROM openjdk:7
                WORKDIR /home
                COPY Main.java  .
                RUN javac /home/Main.java
                CMD [""java"", ""Main""]",
        _ => $@"
                FROM python
                WORKDIR /home
                COPY {fileName} .
                CMD python /home/{fileName}"
    };
    using var dockerFileStream = File.Create(dockerFilePath);
    var dockerFileBytes = new UTF8Encoding(true).GetBytes(dockerFile);
    dockerFileStream.Write(dockerFileBytes, 0, dockerFileBytes.Length);
}
```

Рис. 3.3. Вміст методу CreateCommandFile

```
/// <summary>
/// Run docker image build script
/// </summary>
/// <param name="directoryPath"></param>
/// <returns>Image id</returns>
1 reference
private string RunDockerImageBuildScript(string directoryPath)
{
    var imageId = Guid.NewGuid().ToString();
    var process = GetProcess($"cd {directoryPath} && docker build -t {imageId} .");
    process.Start();
    process.WaitForExit();
    return imageId;
}
```

Рис. 3.4. Вміст методу RunDockerImageBuildScript

RunDockerContainer (рис. 3.5) – метод, який створює контейнер та отримує результат виконання коду надісланого в запиті. Приймає imageId – ім’я, або ідентифікатор образу створеного в попередньому методі. Для запуску контейнеру були використані флаг -m, що забезпечує запуск контейнеру з лімітованої кількістю пам’яті та параметр timeout для

максимального часу життя контейнеру. Їх використання забезпечує безпеку від infinite loop та коду, що споживає велику кількість пам'яті, що може перенавантажити сервер, тобто надає стандартизацію та певний мікроконтроль над використанням ресурсів системи одним контейнером.

```
/// <summary>
/// Run docker container
/// </summary>
/// <param name="imageId"></param>
/// <returns>Image id</returns>
1 reference
private string RunDockerContainer(string imageId)
{
    var timeout = new TimeSpan(0,0, 20);
    var task = Task.Run(() =>
    {
        var process = GetProcess($"docker run --rm -m 100M --name {imageId} {imageId}");
        process.Start();
        var result = process.StandardOutput.ReadToEnd();
        var errorResult = process.StandardError.ReadToEnd();
        process.WaitForExit();
        return $"{result}{errorResult}";
    });
    if (task.Wait(timeout))
        return task.Result;
    var containerStopProcess = GetProcess($"docker container stop {imageId}");
    containerStopProcess.Start();
    containerStopProcess.WaitForExit();
    return "Timed out";
}
```

Рис. 3.5. Вміст методу RunDockerContainer

RemoveDockerImage (рис. 3.6) – це метод, який слугує для видалення створеного образу після закінчення виконання програми та повернення результату, приймає imageId відповідний до непотрібного образу.

```
/// <summary>
/// Remove docker image
/// </summary>
/// <param name="imageId"></param>
/// <returns>Image id</returns>
1 reference
private void RemoveDockerImage(string imageId)
{
    var process = GetProcess($"docker rmi {imageId}");
    process.Start();
    process.WaitForExit();
}
```

Рис. 3.6. Вміст методу RemoveDockerImage

GetProcess (рис. 3.7) – метод, який забезпечує запуск команд у терміналі та перенаправляє вивід StandardOutput, StandardError до об’єкту Process. Приймає command – команда, яку необхідно виконати у терміналі.

```
4 references
private Process GetProcess(string command) =>
    new Process
    {
        StartInfo = new ProcessStartInfo
        {
            FileName = "/bin/bash",
            Arguments = $"-c \"{command}\"",
            RedirectStandardOutput = true,
            RedirectStandardError = true,
            UseShellExecute = false,
            CreateNoWindow = true,
        }
    };
```

Рис. 3.7. Вміст методу GetProcess

Run (рис. 3.8) – метод, необхідний для правильного порядку виклику попередньо вказаних методів та забезпечує принцип інкапсуляції.

```
3 references
public string Run(string command, LanguageType languageId)
{
    var (directoryPath, fileName) = CreateCommandFile(command, languageId);
    CreateDockerFile(directoryPath, fileName, languageId);
    var imageId = RunDockerImageBuildScript(directoryPath);
    var result = RunDockerContainer(imageId);
    RemoveDockerImage(imageId);
    Directory.Delete(directoryPath, true);
    return result;
}
```

Рис.3.8. Вміст методу Run

Висновки до розділу 3

В даному розділі була описана загальна структура розробленої системи безпечного завантаження та виконання коду на сервері. Система написана на мові C# використовуючи фреймворк ASP .NET Core з використанням технології docker для інкапсуляції виконуваного коду.

Основою системи являються два класи CodeController та CodeService, перший виконує роль приймача запитів та корегування маршрутизації в системі, другий містить основні рішення системи. Також створений додатковий тип LanguageType, який є типом перелічування enum, та слугує для ініціалізації мов програмування та клас CodeModel, який містить опис об’єкту запиту.

РОЗДІЛ 4

РЕЗУЛЬТАТИ РОЗРОБКИ

У даному розділі описується робота системи безпечного завантаження та виконання коду на сервері. Система була розроблена на основі технології docker, а також з використанням технологій для створення інтерфейсу користувача, для наглядної демонстрації можливого використання розробки на веб-сторінках для реалізації функціональності компіляторів, або IDE у веб-додатках. Демонстрація роботи розробленої системи розділена на дві частини, а саме на те, що відбувається на сервері при надсиланні на нього запиту, та демонстрація взаємодії користувача із інтерфейсом веб-сторінки.

4.1. Демонстрація роботи системи

Якщо сервер доступний для використання, тоді для перевірки його працездатності можна використати будь яку систему для тестування API. Для демонстрації роботи системи було обрано використання безкоштовного сервісу Postman.

Для початку роботи системи сервер має отримати запит, на рис. 4.1 показано приклад надісланого запиту з кодом «print(“Hello world”)» на мові Python через використання інструментарію Postman.

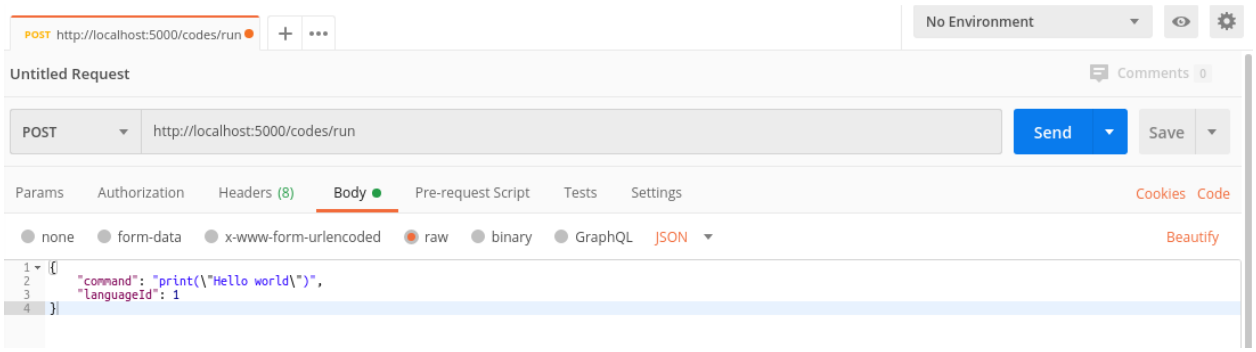


Рис. 4.1. Відправлення запиту на сервер через сервіс Postman

На вхід сервер повинен прийняти два необхідних параметри, перший це код програми (command), яку користувач хоче протестувати та другий –

ідентифікатор мови програмування (languageId), на якій був написаний код. Система працює з трьома мовами, кожна з яких має свій ID, для Python це 1, для C# – 2 і для Java – 3. Після того, як сервер приймає запит, на ньому створюється директорія з випадковою назвою (рис. 4.2) та файлом з кодом надісланим в запиті, та відповідно до мови програмування вказаної в запиті (рис. 4.3).

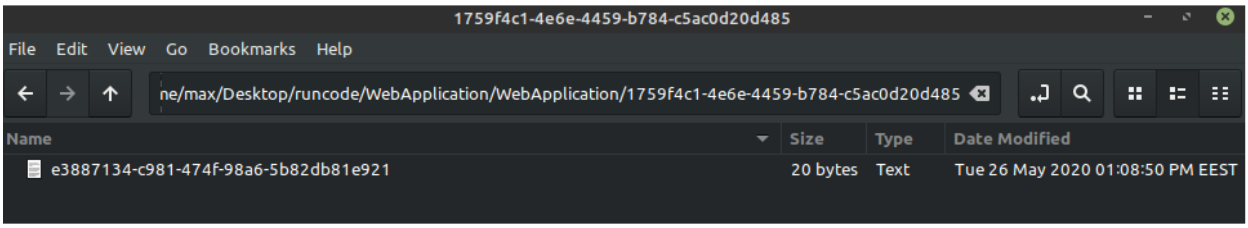


Рис. 4.2. Створення директорії з кодом на сервері

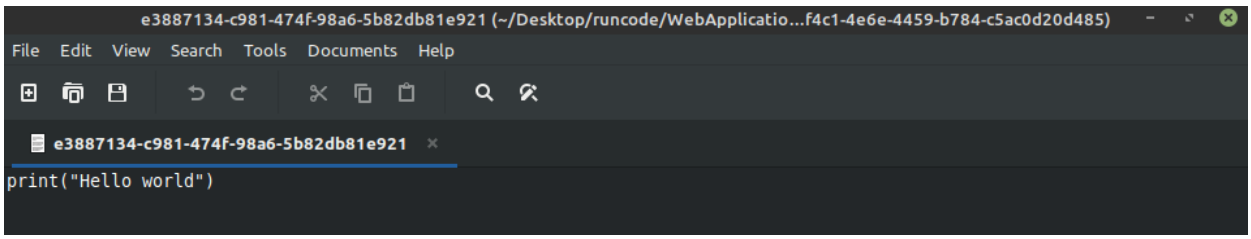


Рис. 4.3. Вміст файлу з кодом

Далі на сервері створюється dockerfile в залежності від обраної мови програмування (рис. 4.5) та додається у створену директорію (рис. 4.4).

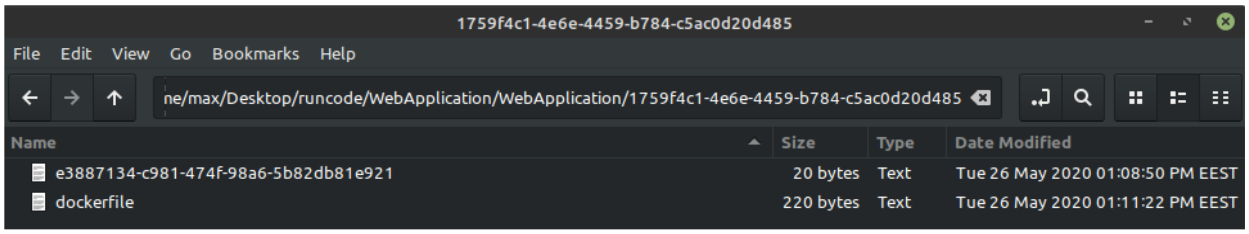


Рис. 4.4. Створення dockerfile у директорії

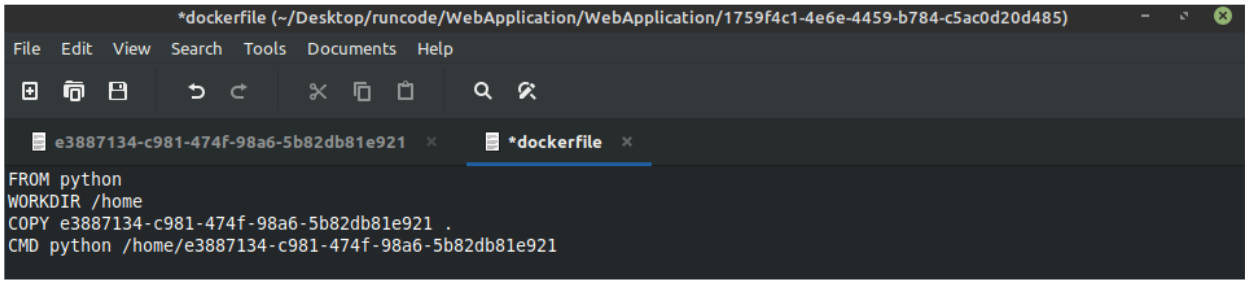


Рис. 4.5. Вміст dockerfile

Наступні перетворення та дії, які відбуваються на сервері показано через термінал командного рядка.

Після створення dockerfile відбувається запуск процесу зборки образу docker, йому надається випадкова назва, а також ідентифікатор (рис. 4.6). Після чого, відповідно до створеного ідентифікатору запускається контейнер (рис. 4.7) з програмою для отримання результату виконання коду.

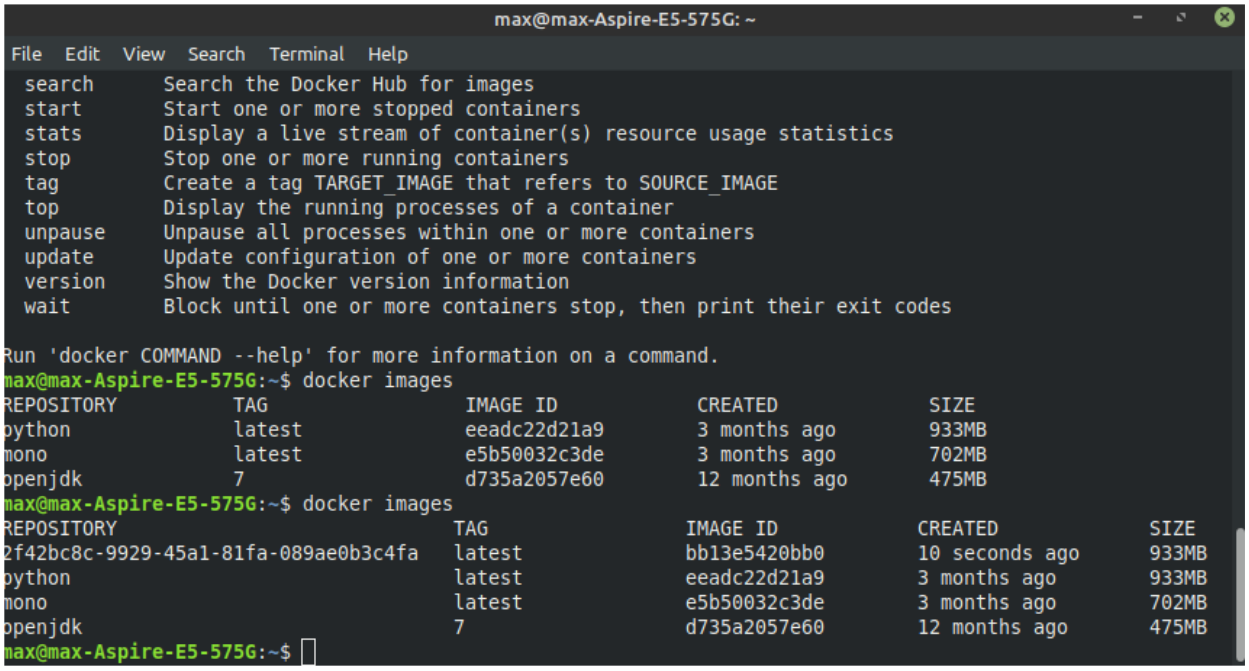


Рис. 4.6. Створення образу docker

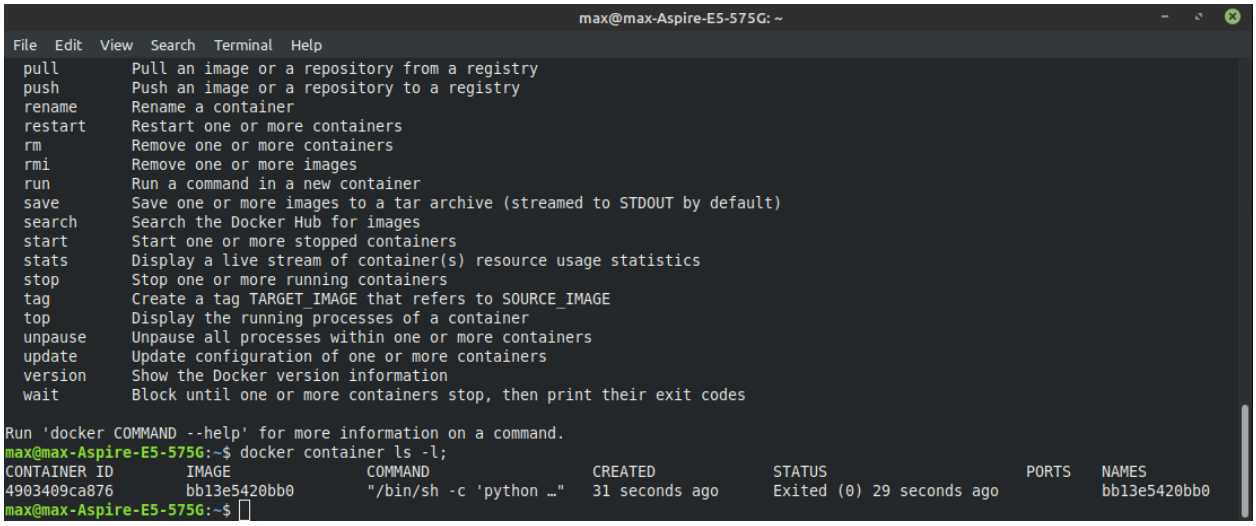


Рис. 4.7. Запуск контейнера для відповідного образу docker

Як тільки програма закінчила обчислення та отримала результат , ці дані передаються на сервер та виконується видалення створеного образу docker (рис. 4.8). Наступними видаляються файл dockerfile та директорія їх знаходження.

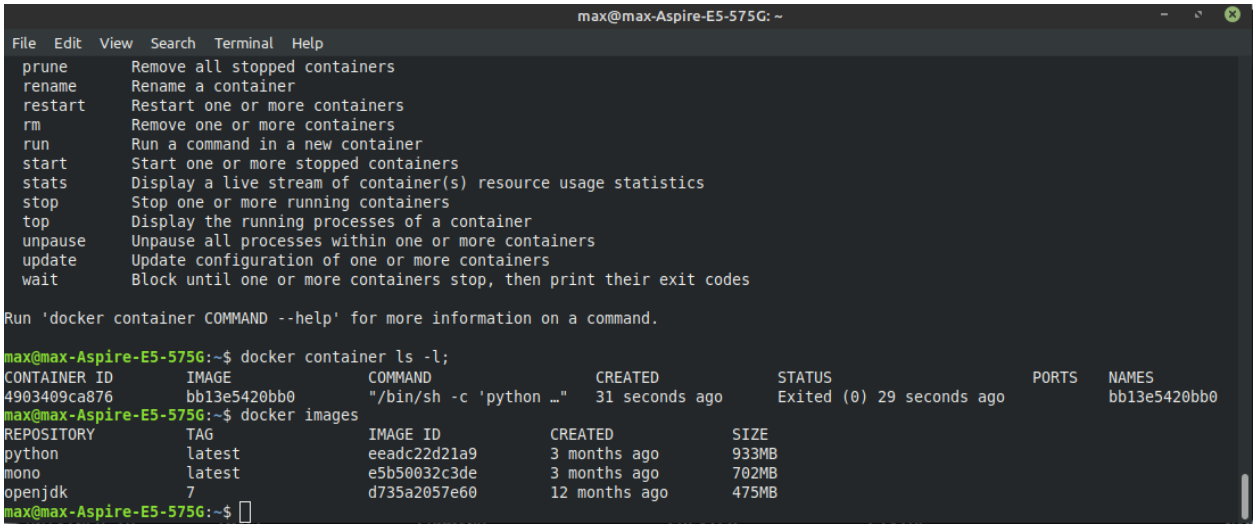


Рис. 4.8. Видалення попередньо створеного образу docker

Після всіх вище перелічених дій на сервіс, з якого був надісланий запит, надсилається результат виконання програми надісланої на сервер (рис. 4.9). В нашому випадку результат прийшов на Postman, з якого і був надісланий, у вигляді рядка «Hello world».

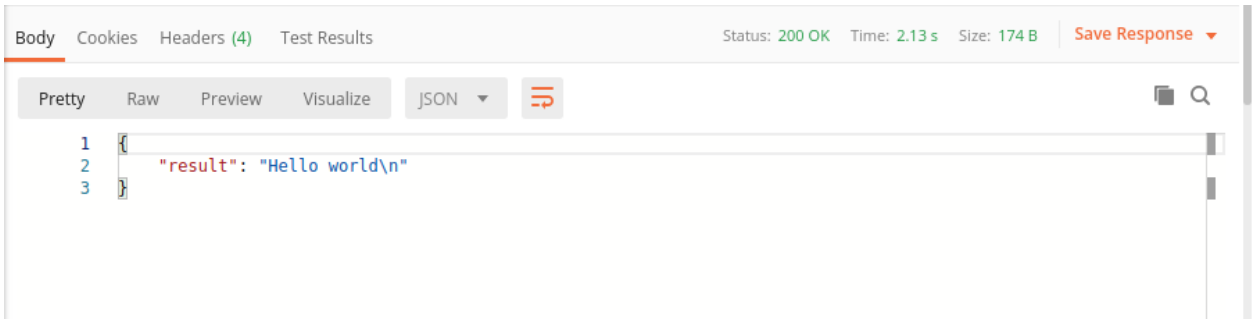


Рис. 4.9. Видалення попередньо створеного образу docker

4.2. Інструкція користувача

Розроблювана система може являтися частиною більш складної структури, програмного забезпечення або веб-сервісу. Такі веб-сервіси називають онлайн-компіляторами, їх користувач не бачить того, як працює

система, а тільки користується функціональними можливостями, які вона надає. Для наглядного прикладу було розроблено інтерфейс веб-сторінки стандартного онлайн-компілятора (рис. 4.10) для надання можливості завантажування та запуску коду. Відповідаючи необхідним параметрам запиту, сторінка містить форму з двома полями для кожного параметру, одне поле через спливаюче меню дає обрати мову програмування на якій буде написаний код, інше ж поле є полем вводу, в яке вписується програмний код.

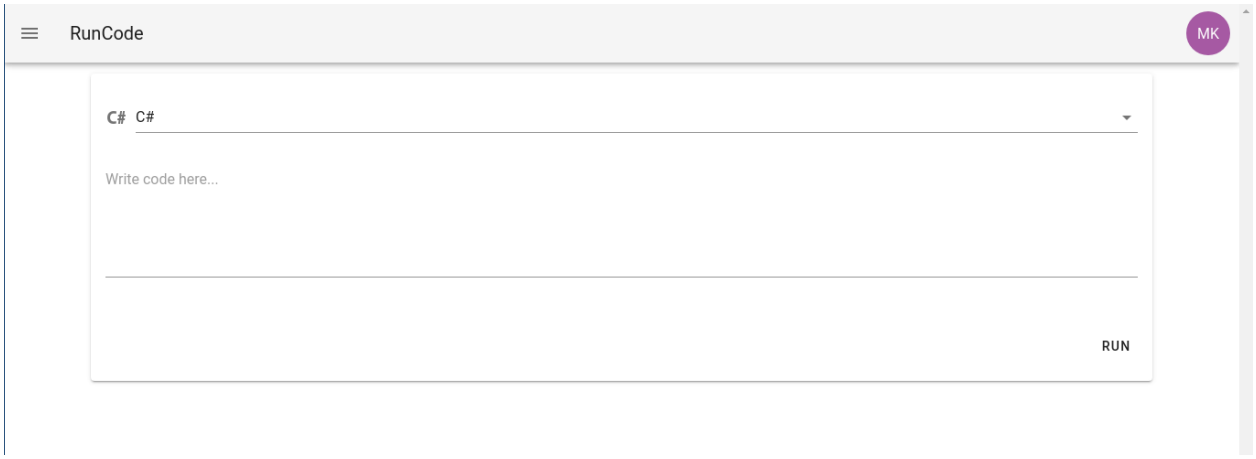


Рис. 4.10. Сторінка з формою для завантаження коду на сервер

Після того, як користувач обрав мову, на якій буде написана програма, та написав відповідний код, йому необхідно натиснути кнопку «run», яка знаходиться у правому нижньому куті форми (див. рис. 4.10). Після цього на сервер буде відправлений запит та почнуть виконуватися всі дії описані в попередньому пункті розділу, а після їх завершення буде отримана відповідь на запит. Коли відповідь буде отримана то користувач побачить відповідне сповіщення та на сторінці з’явиться поле «Result», у якому відобразиться результат виконання коду (рис. 4.11), або вивід виключення та помилки при їх допущенні в програмі користувача (рис. 4.12).

Рис. 4.11. Позитивний результат виконання коду

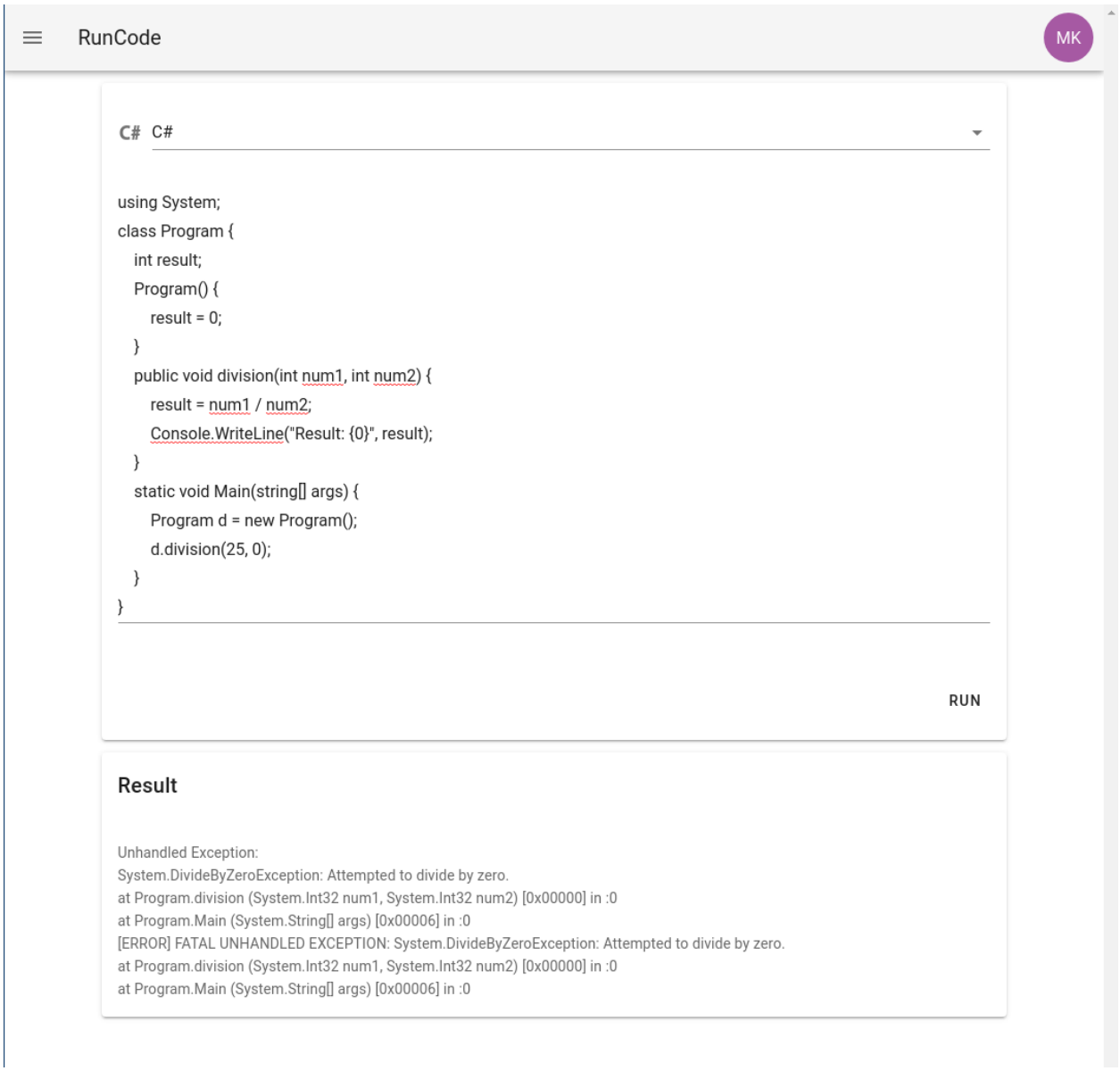


Рис. 4.12. Вивід виключення, як результат виконання коду з помилкою

Висновки до розділу 4

В даному розділі була розглянута робота системи безпечного завантаження та виконання коду на сервері. Розділ поділений на дві частини, перша показує, як сервер реагує на розроблену систему, та показує виконання створених рішень, які були описані в попередньому розділі, друга частина показує інструкцію для користувача, який використовує систему як засіб отримання відповіді. Правильна робота всіх модулів, забезпечила вдалий результат, що вказує на вірність прийнятих рішень.

Розроблена система показала успішні результати при тестуванні, а також вдалий вибір технології. Написана програма працює стабільно та виконує всі поставлені завдання. Також розроблений інтерфейс може бути підґрунтям для впровадження технології в інші сервіси.

ВИСНОВКИ

Ціллю даної дипломної роботи була реалізація системи безпечного завантаження та виконання коду на сервері. Основним завданням було забезпечити безпеку серверу від навмисних та ненавмисних атак з боку користувачів. Під час роботи над проектом було розглянуто та проаналізовано велику кількість теоретичного матеріалу стосовно теми, а також проведений огляд існуючих аналогів систем та технологій для рішення поставленої задачі. Після завершення роботи над дипломним проектом було зроблено декілька висновків:

1. Головною проблемою з якою зустрічаються розробники систем для запуску коду на сервері є саме безпека серверної частини. Зумовлено це тим, що програми користувачів системи можуть містити код, який може доступатися до файлової системи серверу, або впливати на процеси в ній, в результаті такі програми можуть створювати незручності в роботі із системою, або нашкодити їй.

2. Для уникнення проблем із запуском коду на сервері, доцільним є використання технологій контейнеризації – розгортання додатків у ізольованих контейнерах системи, що дають можливість обмежувати надання ресурсів системи для кожної окремої програми та забезпечує безпеку для основної системи хоста.

3. Серед наявних на ринку технологій контейнеризації найбільш популярною і розвиненою є технологія Docker. Серед інших технологій вона є, певно, найрозвиненішою та надає можливості для створення легких контейнерів з розгортанням окремих додатків, що зменшує потреби в ресурсах та надає гнучкості в розробках. Також, дана технологія являється відкритою для користування, що дозволяє розроблювати систему без грошових витрат.

4. При роботі з технологією Docker доцільно використовувати Dockerfile для автоматизації побудови образів та створення контейнерів.

Докладна документація та розвинута система команд та інструкцій, дають можливість без перешкод автоматизувати систему розгортання додатків для запуску коду в контейнерах.

5. Використовуючи фреймворк ASP .NET Core та технологію Docker, була розроблена система безпечного завантаження та виконання коду на сервері. В запиті на сервер вказується два параметри: код програми та мова програмування, а у відповідь отримується результат виконання програми. Розроблена система забезпечує надання всіх необхідних можливостей користувачу та безпеки серверу на якому програма запускається.

6. Створено веб-додаток з закінченим інтерфейсом. На даному етапі вже можливе підключення такого додатку до існуючих для розширення їх можливостей.

ПЕРЕЛІК ПОСИЛАНЬ

1. Linux containers [Електронний ресурс] – Режим доступу до ресурсу: https://linuxcontainers.org/lxd/introduction/.
2. Авторские статьи об OpenSource [Електронний ресурс] – Режим доступу до ресурсу: http://vasilisc.com/lxd-2-0-introduction-to-lxd.
3. Solaris Container (Solaris Zone) [Електронний ресурс] – Режим доступу до ресурсу: https://ru.bmstu.wiki/Solaris_Containers.
4. Solaris Containers: What You Need to Know [Електронний ресурс] – Режим доступу до ресурсу: https://www.sumologic.com/blog/solaris-containers-need-know/.
5. Проєкт CoreOS представив Rocket [Електронний ресурс] – Режим доступу до ресурсу: https://www.opennet.ru/opennews/art.shtml?num=41168
6. Випуск rkt 1.0 [Електронний ресурс] – Режим доступу до ресурсу: https://www.opennet.ru/opennews/art.shtml?num=43824
7. Изучаем Docker, часть 1: основы [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/company/ruvds/blog/438796/.
8. Полное практическое руководство по Docker: с нуля до кластера на AWS [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/310460/.
9. Docker. Зачем и как [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/309556/.
10. View logs for a container or service [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/config/containers/logging/.
11. Best practices for writing Dockerfiles [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.
12. Dockerfile reference [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/engine/reference/builder/.

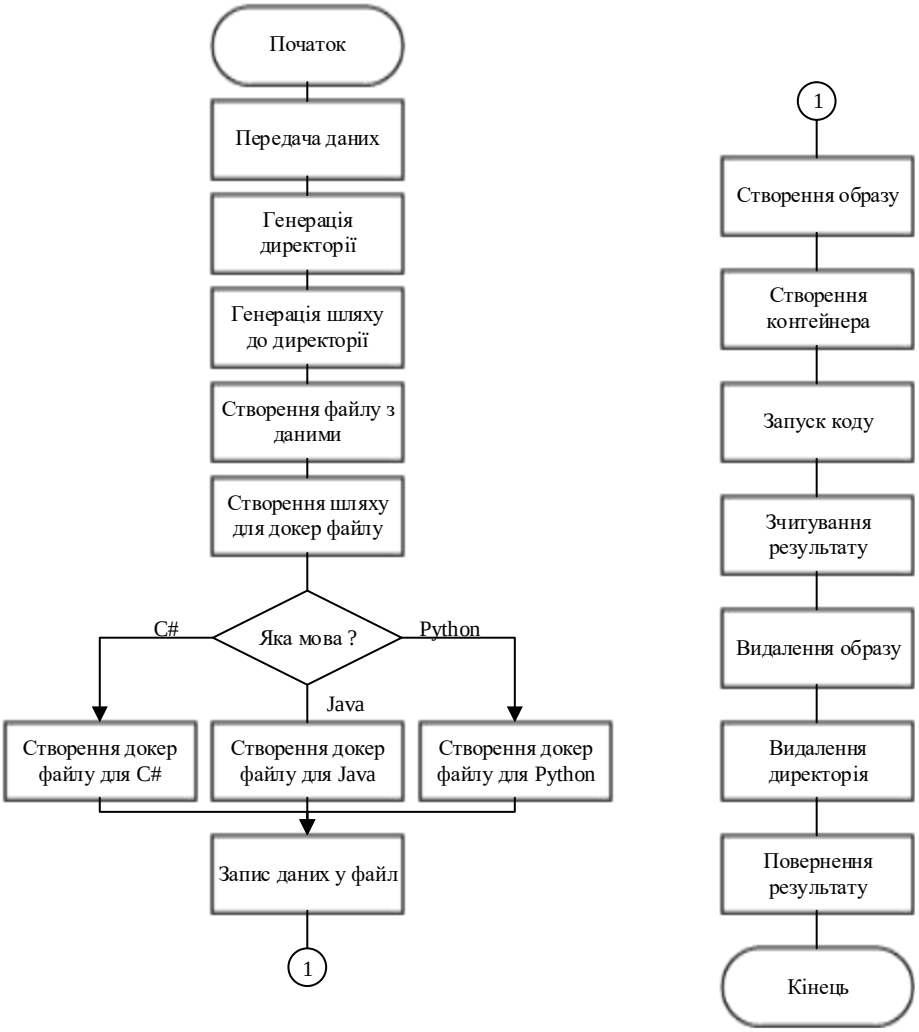
13. Docker run reference [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/engine/reference/run/#runtime-constraints-on-resources.
14. Runtime options with Memory, CPUs, and GPUs [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/config/containers/resource_constraints/#--memory-swappiness-details.
15. Docker stop [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/engine/reference/commandline/stop/#examples.
16. Docker rmi [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/engine/reference/commandline/rmi/.
17. Docker images [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/engine/reference/commandline/images/.
18. Руководство по ASP.NET Core 3 [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/aspnet5/.

Додаток А

Принципова схема алгоритму завантаження та виконання коду на сервері

до дипломного проєкту
на тему: «Система безпечного завантаження та виконання коду на сервері»

Київ – 2020 року



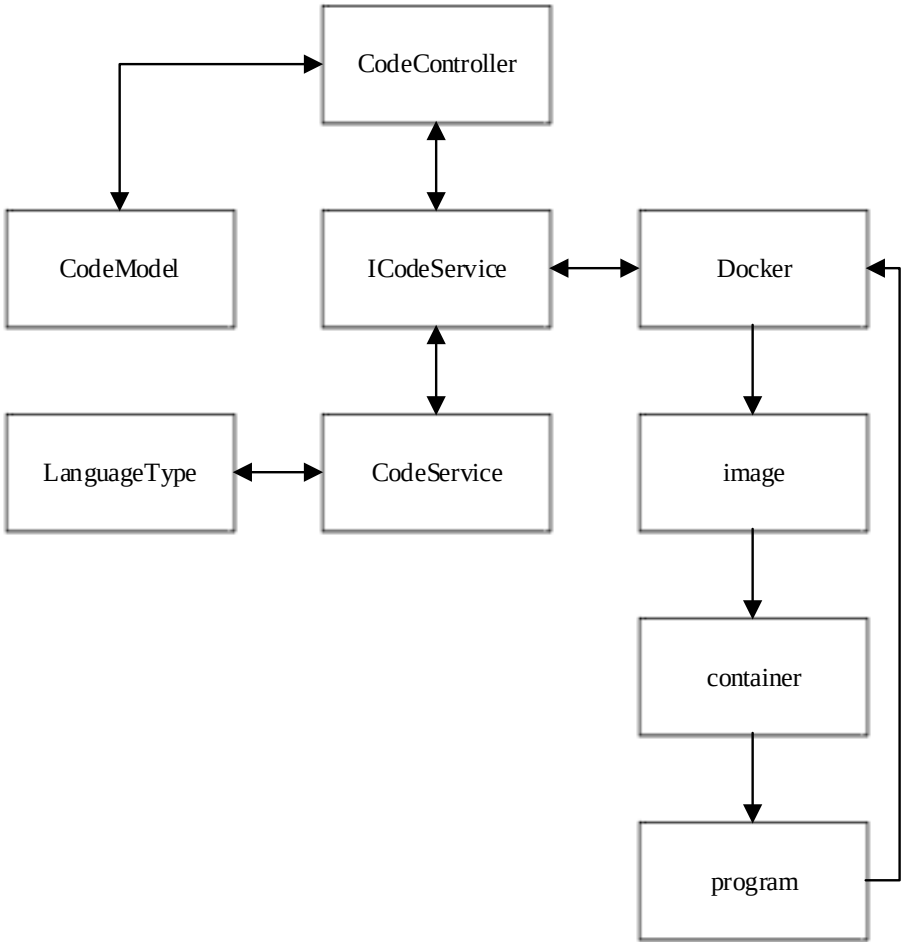
					ІАЛЦ.467100.004 Д1		
					Система безпечного завантаження та виконання коду на сервері Принципова схема алгоритму завантаження та виконання коду на сервері	*Літ.*	*Маса.*	*Масш.*
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*				
Розроб.		*Ханін М. Є.*						
Перевір.		*Новотарський М.*				*Аркуш* 1	*Аркушів* 1	
Т. Конто.								
						НТУУ «КПІ», ФІОТ ІО-64		
Н. Контр.		*Сімоненко В.П.*						
Затверд.								

Додаток Б

Структурна схема системи завантаження та виконання коду на сервері

до дипломного проєкту
на тему: «Система безпечного завантаження та виконання коду на сервері»

Київ – 2020 року



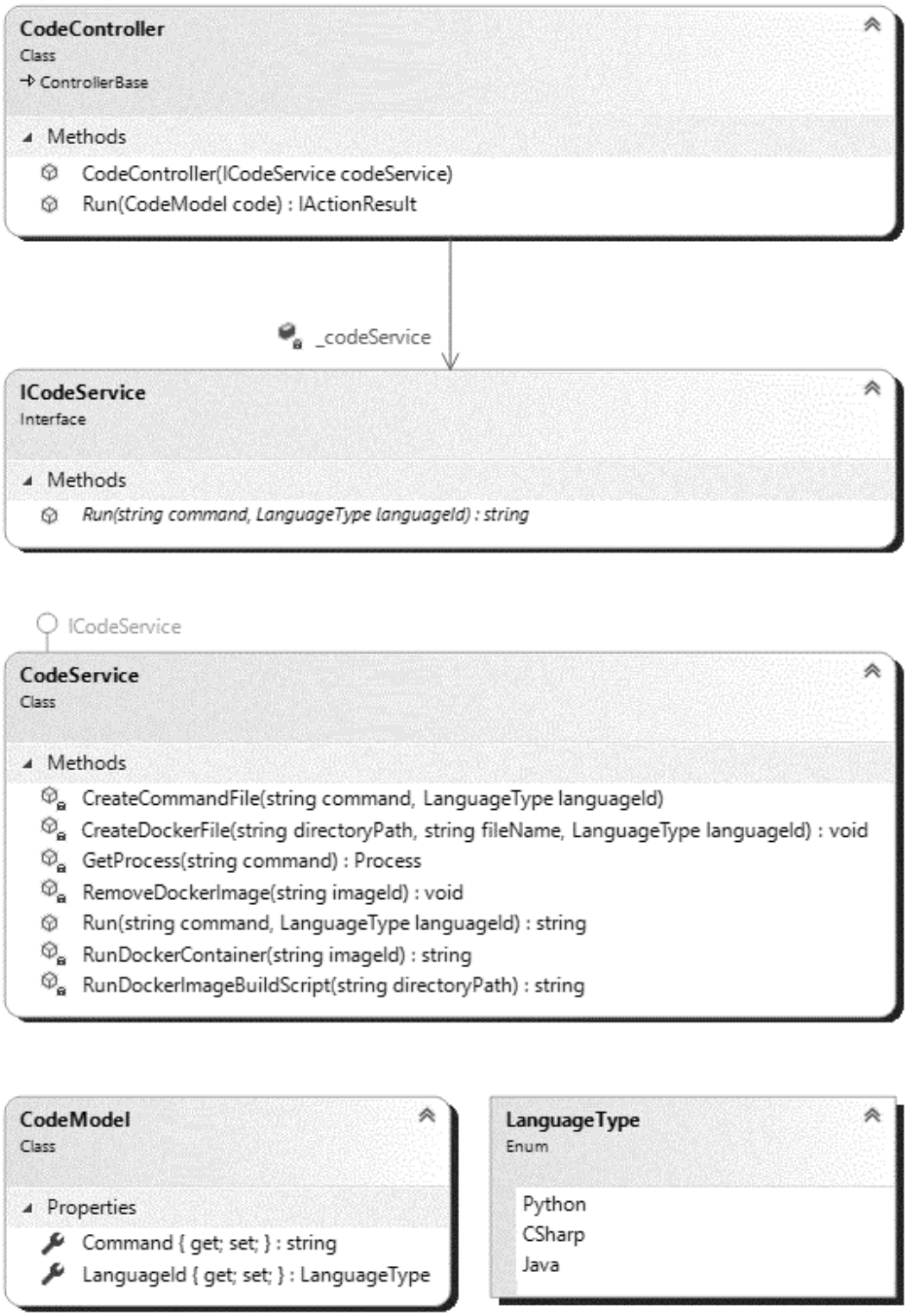
					ІАЛЦ.467100.005 Д2		
					Система безпечного завантаження та виконання коду на сервері Структурна схема системи завантаження та виконання коду на сервері	Літ.	Маса.	Масш.
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		Ханін М. Є.				Аркуш 1	Аркушів 1	
Перевір.		Новотарський М.						
Т. Конто.								
Н. Контр.		Сімоненко В.П.				НТУУ «КПІ», ФІОТ ІО-64		
Затверд.								

Додаток В

Функціональна схема класів системи завантаження та виконання коду на сервері

до дипломного проєкту
на тему: «Система безпечного завантаження та виконання коду на сервері»

Київ – 2020 року



					ІАЛЦ.467100.006 ДЗ		
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*	*Система безпечного завантаження та виконання коду на сервері Функціональна схема класів системи завантаження та виконання коду на сервері*	*Літ.*	*Маса.*	*Масш.*
Розроб.		*Ханін М. Є.*						
Перевір.		*Новотарський М.*				*Аркуш* 1	*Аркушів* 1	
Т. Конто.								
Н. Контр.		*Сімоненко В.П.*				*НТУУ «КПІ», ФІОТ ІО-64*		
Затверд.								